CHAPTER

1 INTRODUCTION

1.1 INTRODUCTION

Programming Languages are notations that are implemented on a machine (computer) for the description of algorithms and data structures. The term **'Programming Language'** is made up of two different words namely **'Programming'** and **'Language'**. These two words are described as :

Programming : When a particular problem is to be solved, it is necessary to design statements or instructions for the computer to carry out. The art of writing instructions for a computer to solve the specific task is known as programming. The concept is illustrated diagrammatically in figure 1.1.

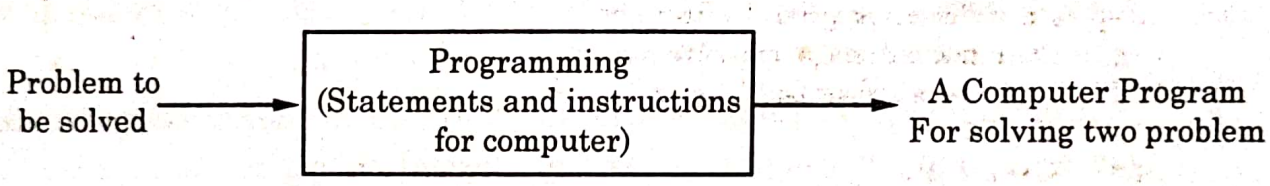


Fig. 1.1. The Concept of Programming.

The output of programming is a well defined set of instructions, called a **program**. A program can be viewed as a function where the output data values are a function of input data values *i.e.*

Output = Program (Input)

Another view of a program is that it models a problem domain and the execution of the program is the simulation of the problem domain *i.e.*

Program = Model of a problem domain

Execution of a program = Simulation of the problem domain

In any case the data objects are central to the programs.

Language : A language is defined as the set of all possible strings, words or sentences that can be derived from a given alphabet (a set of input symbols denoted by Σ

Mathematically, a language L is defined as :

$$L \subseteq \sum^*$$

where $\sum^*$= Set of the possible strings derived from a given alphabet $\sum$ *e.g.* consider a machine language where $\sum = \{0, 1\}$

Therefore $\sum^* = \{ \wedge, 0, 1, 00, 01, 10, 11, \ldots\ldots \}$

Clearly $L \subseteq \sum^*$.

Another example of a language is English language (a natural language) *i.e.* a set of all possible words, sentences or strings derived from the alphabet.

$$\sum = \{a, b, c, \ldots\ldots, z\}$$

Now, based on the above description of terms **'Programming'** and **'Language'** the term **'Programming Language'** is defined by different authors. All these definitions convey the same meaning. These definitions are given as.

- A programming language is a stylized communication technique intended to be used for controlling the behaviour of a machine (often a computer). Like human languages, the programming languages have syntactic and semantic rules used to define structure and meaning.
- A programming language is a formal notation for describing algorithms for execution by computer. Like all formal notations, a programming language has three major components ; syntax, semantics and pragmatics.
- A Programming language is a notation for writing computer programs used for specifying, organizing and reasoning about computations. The three basic components of a programming language are syntax, semantics and pragmatics.
- A programming language is an artificial language that can be used to control the behaviour of a machine, particularly a computer. Programming languages, like natural languages are defined by syntactic and semantic rules which describe their structure and meaning respectively.

Programming languages are used to facilitate communication about the task of organizing, manipulating information and to express algorithms precisely. Syntax, semantics and pragmatics are the components forming the base of a programming language.

1.1.1 DESCRIPTION OF PROGRAMMING LANGUAGES

From the above definitions of **'Programming Language'** it is clear that a complete description of programming language includes the computational model (*i.e.* a collection of values and operations), the syntax and semantics of programs, and the pragmatic considerations that shape the language. Here we provide a brief description of syntax, semantics and pragmatics of a programming language.

Syntax :- The syntax of a programming language refers to the structure or form of programs.

Semantics :- The semantics of a programming language describes the relationship between a program and the model of computation.

Pragmatics :- The pragmatics of a programming language describes the degree of success with which a programming language meets its goals both in its faithfullness to the underlying model of computation and in its utility for human programmers.

1.1.2 PROGRAMMING LANGUAGES AND NATURAL LANGUAGES

The similarities between a programming language and a natural language are as given below :

(1) Both Programming languages and Natural languages provides a means of communication.

(2) Both Programming languages and Natural languages are defined by syntactic, semantic and pragmatic rules.

However, the Programming languages are different from natural languages in that

(1) Unlike natural languages, programming languages are a formal notation.

(2) Unlike natural languages, programming languages are used to control the behaviour of a machine, particularly a computer.

(3) Unlike natural languages, the programming languages require a greater degree of precision and completeness.

(4) While communicating using natural languages, the human authors and speakers can be ambiguous and containing some errors but still their intent is understood. However, the computers do exactly what they are told to do and cannot understand the code the programmer "intended" to write.

(5) Unlike natural languages, some programming languages are used by one device to control another. For example, **Postscript** programs are frequently created by another program to control a computer printer or display.

1.2 WHY WE STUDY THE CONCEPTS OF PROGRAMMING LANGUAGES

As described in earlier section, a programming language is a formal notation for describing algorithms for execution by computer. The widespread use of programming languages began with the arrival of Fortran in 1957. Hundreds of different programming languages have been designed and implemented. Even in 1969, Sammet listed 120 that were fairly widely used, and many others have been developed since then. Many programmers confine their programming entirely to one or two programming languages like C, C++, Ada or FORTRAN. Here, in this section we discuss some reason why computer science students and professional software developers should study wide variety of general programming language-design and evaluation concepts. This discussion is valuable for those who believe that a working knowledge of one or two programming languages is sufficient for computer scientists. The main reasons to study different kinds of programming languages are:–

1. **To improve your ability to develop effective algorithms :–** Many languages provide features that when used properly are of benefit to the programmer but when used improperly may waste large amounts of computer time or lead the programmer into time consuming logical errors. For example, recursion, when properly used allows the direct implementation of elegant and efficient algorithms. But used improperly, it may cause as astronomical increase in execution time. A basic knowledge of its principles and implementation techniques allows the programmer to understand the relative cost of recursion in a particular language and from this understanding we determine whether its use is warranted in a particular programming situation.
2. **To improve your use of your existing programming language :–** By understanding how features in the language are implemented, it greatly increases our ability to write efficient programs.
3. **To increase your vocabulary of useful programming constructs :–** It is widely believed that the depth at which we can think is influenced by the expressive power of the language in which we communicate our thoughts. Languages serves both as an aid and a constraint to thinking. People use languages to express thoughts, but languages serves also to structure how one thinks, to one extent that it is difficult to think it ways that allow no direct expression in words. Familiarity with a single programming language tends to have similar constraining effect. In searching for data and program structure suitable to the solution of a problem, one tends to think only of structures immediately expressible in the languages, with which one is familiar. By studying the wide range of programming language constructs and their implementation a programmer increases his/ her programming **"vocabulary".** For example, the subprogram control structure known as co-routines is useful in many programs, but few languages provide a co-routine feature directly.
4. **To allow a better choice of Programming language :–** Many professional programmers have had a little formal education in computer science; rather, they have learned programming on their own or through in-house training programs. Such training programs often teach one or two languages that are directly relevant to the current work of the organization. When the situation arises, a knowledge of a variety of languages may allow choice of just the right language. For a particular project, thereby reducing the required coding effort. *e.g.*
   - Applications requiring numerical calculations can be designed using the languages like C, FORTRAN and ADA.
   - Applications useful in decision making (*i.e.* Artificial Intelligence applications) can be designed using the languages like LISP, ML or PROLOG.
5. **To make it easier to learn a new language :–** Computer programming is a young discipline, and design methodologies, software development tools, and programming language are still in a state of continuous evolution. This makes software development an exciting profession, but it also means that continuous learning is essential. A thorough knowledge of variety of programming constructs and implementation techniques allows the programmer to learn a new programming language more easily.
6. **To make it easier to design a new language :–** To a student, the possibility of being required at some future time to design a new programming language may seem remote. However, most professional programmers occasionally do design languages of one sort or another. For example, most of a software systems require the user to interact in some way, even if only to enter data and commands. In simple situations, only a few data values are entered, and the input format language is trivial. On the other hand, the user might be required to traverse several levels of menus and enter a variety of commands, as in the case of a word

processor. In such systems, the user interface is a complex design problem. A critical examination of programming languages, will help in the design of such complex systems and more commonly it will help users to examine and evaluate such products.

7. **Overall advancement of computing :-** Finally, there is a global view of computing that can justify the study of programming language concepts. Although, it is usually possible to determine why a particular programming language became popular, it is not always clear, at least in retrospect, that the most popular languages are the best available. In some cases, it might be concluded, a language became widely used, at least in part, because those in positions to choose languages were not sufficiently familiar with programming language concepts.

1.3 A BRIEF HISTORY OF PROGRAMMING LANGUAGES

The first programming language predate the computer. From the first, the languages were codes. During a nine-month period in 1842-1843, **Ada Lovelace** specified a detailed method for calculating Bernoulli numbers with Charles Babbage's Analytical Engine. Some historians recognized it as the world's first computer program.

After some years, Herman Hollerith realized that he could encode the information on **punch cards.** He then proceeded to encode the 1890 census data on punch cards. The first computer codes were specialized for the applications. In the first decades of the twentieth century, numerical calculations were based on decimal numbers. Eventually it was realized that logic could be represented with numbers, as well as with words.

Like many "firsts" in history, the first modern programming language is hard to identify. From the start, the restrictions of the hardware defined the language. Punched cards allowed so columns, but some of the columns had to be used for a sorting number on each card. FORTRAN included some keywords which were the same as English words such as "IF", "GOTO" and "CONTINUE". The use of magnetic drum for memory meant that computer programs also had to be interleaved with the rotations of the drum. Thus the programs were more hardware dependent than today.

To some people the answer depends on how much power and human-readability is required before the status of "programming language" is granted. **Jacquard looms** and **Charles Babbage's Difference Engine** both had simple,

extremely limited languages for describing the actions that these machines should perform. One can even regard the punch holes on a **player piano** scroll as a limited **domain-specific programming language,** albeit not designed for human consumption.

In the 1940s the first recognizably modern, electrically powered computers were created. The limited speed and memory capacity forced programmers to write hand tuned **assembly language** programs. In 1948, **Konrad Zuse** published a paper about his programming language **Plankalkul.** However, it was not implemented in his time and his original contributions were isolated from other developments. Some important languages, that were developed in this period includes Plankalkul (Konrad Zuse), ENIAC coding system and C – 10.

In the 1950s the first three modern programming languages, whose descendants are still in widespread use today were designed.

FORTRAN, the **Formula Translator**,, invented by John W. Backus et al ;

LISP, the **LIST Processor**, invented by John McCarthy et al ;

COBOL the **Common Business Oriented Language**;, created by the short range Committee, heavily influenced by Grace Hopper.

Some other programming languages, developed during the 1950s and 1960s include Regional Assembly Language, Autocode, ALGOL58, APL, Simula, BASIC and PL/I. The period from the late 1960s to the late 1970s brought a major flowering of programming languages. Most of the major language paradigms now in use we are invented in this period. Some important languages developed during this period include, simula, small talk, Pascal, Forth, C, Prolog. ML, SQL. The 1980s were years of relative consolidation. One important new trend in language design was an increased focus or programming for large-scale systems through the use of modules, or large-scale organizational units of code. Modula, Ada and ML all developed notable module systems in the 1980s. Some other important languages that were developed during 1980's include C ++, Eiffel, Perl and FL (Backus). The rapid growth of the internet in the mid-1990s was the next major historic event in programming languages. Some important languages that were developed in this period includes Haskell, Python, Java, Ruby, PHP and C #. Programming language evolution continues, in both industry and research. Some current directions.

- Mechanical for adding security and reliability verification to the language.
- Alternative mechanisms for modularity, mixing, delegates, aspects.
- Component-oriented software development.
- Metaprogramming, reflection or access to the abstract syntax tree.

- Increased emphasison distribution and mobility.
- Integration with databases, including XML and relational databases.
- Open source as a developmental philosophy for languages.
- Support for Unicode.

A brief history of different programming languages is shown in Table 1.1 below:

Name	Date	Origin of name	A brief description
Fortran	1957	Formula Translation	The first programming language ever used was Fortran-66 – a simple and efficient language for numerical computations, traditionally used by engineers.
Lisp	1959	List Processing	Favoured by academics, used in artificial intelligence research.
COBOL	1959	Common Business Oriented Language	A wordy language traditionally used by companies for boring programs that shuffle employee records around.
BASIC	1964	Beginners All-purpose Symbolic Instruction Code	BASIC was intended as a toy language to introduce students to programming, after which they could go on and do real programming in more capable languages.
PL/1	1965	Programming Language 1	An attempt to assemble the best features of Fortran, COBOL, and Algol into one coherent language.
APL	1968	A Programming Language	A bizarre and cryptic language using mathematical-style notation ; requires a special character set.
Forth	1970	"Fourth-generation" in five letters	A truly oddball language created by an intelligent eccentric. Very simple and flexible, but perhaps impractical for most people.
Prolog	1972	Programming in Logic	Probably used mainly by specialists in logic and artificial intelligence.

Name	Date	Origin of name	A brief description
Pascal	1974	Blaise Pascal, French 17th-century scientists	Pascal is superficially similar to Algol-60, attractive to read, but hampered by inflexible type restrictions.
C	1978	Successor to B	C is another language descended from Algol-60, but its copious use of symbols and its cryptic vocabulary give it an ugly and unfriendly appearance compared with Pascal. Probably because it was used to write Unix, and Unix has been widespread in universities for a long time, it became the favourite language of professional programmers.
Modula-2	1980	Pascal with modules	Intended by Niklaus wirth as a successor to Pascal, but never achieved Pascal's popularity.
Smalltalk	1983	Originally intended for use by children	A pure object-oriented language that is simple, elegant, and original, but spoiled by an over-complex class library.
C++	1985	Incremented	This just adds object-oriented extensions to the older C language. It retains most of the advantages and disadvantages of C, but it's conceptually more complex than C and therefore more difficult to learn and use.
Eiffel	1986	Gustave Eiffel, French engineer	An object-oriented language created by two Frenchmen and apparently intended for use by software engineers in large companies.

Name	Date	Origin of name	A brief description
Ada	1987	Augusta Ada King (1815-1852), Countess of Lovelace, the world's first programmer.	Pascal-based language originally created in 1983 for the exclusive use of the U.S. Department of Defense. Made available to the public in 1987, but doesn't seem to have become popular.
Perl	1987	Practical Extraction and Report Language	I've used Perl to write a few CGI scripts for Web sites at work.
Visual Basic	1987	BASIC with a visual development environment.	Windows-only, Brings BASIC into the age of the graphical user interface.
Modula-3	989	A modular language newer than Modula-2	One of those languages that seems to take a feature or two from every previous language. It was a strange decision to call it Modula-3, considering that it wasn't designed by Niklaus Wirth and isn't compatible with Modula-2.
ICL	1989	Tool command language	A scripting language intended to be used in combination with other programming languages.
Python	1991	Monty Python's	An attractively simple (and free !)
		Flying Circus	Interpreted language with good basic facilities, which runs on almost any computer once you install the interpreter.
Java	1994	Coffee	Intended as a simplified and modernized version of C++. It does at least seem an improvement on C++. It's becoming very popular, and it's very portable from one platform to another, which is of course an asset. However, it's

Name	Date	Origin of name	A brief description
			normally implemented as a semi-interpreted language, so the user needs to install an interpreter in order to run Java programs. Some Web browsers include a Java interpreter.
Delphi	1995	Ancient Greek town	Windows-only. Delphi is really just Turbo Pascal with object-oriented extensions and a modern visual development environment. It's easier than C++ and more powerful than Visual Basic.

Table 1.1 : A brief description of some programming languages.

We briefly summarize language development during the early days computing, generally from the mid-1950s to the early 1970s.

- **Numerically based languages :-** Early computer technology dates from the era just before world war II through the early 1940's. In the early 1950's, symbolic notations started to appear. Grace Hopper lead a group at Univac to develop the A-O languages, and John Backus developed speed coding for the IBM-701.

  The real break through occurred in 1957 when Backus led a team to develop FORTRAN (**Formula Translator**). The FORTRAN data were oriented around numerical calculations, but the goal was a full fledged programming language including control structures, conditionals, input and output statements. FORTRAN was extremely successful and changed the programming forever. FORTRAN was revised as FORTRAN II in 1958 and FORTRAN IV a few years later. Finally in 1966, FORTRAN IV became a standard under the name FORTRAN 66 and has been upgraded twice since, to FORTRAN 77 and FORTRAN 90.

  The success of FORTRAN caused fear in Europe. Therefore GAMM (The German society-of applied Mathematics) organized a committee to design a universal language. In US, the ACM (Association for computing Machinery) also organized a similar committed. These two committees merged under the leadership of Peter-Naur and developed the IAL

(International Arithmetic language). The common usage of IAL forced the official name change and the language become known as ALGOL 58 which was later revised as ALGOL 60. The goals for ALGOL notation were.

- close to standard mathematics and useful for description of algorithms.
- Not bound to a single computer architecture.
- Compatible into machine language.

ALGOL never achieved commercial success in the U.S., although it did achieved some success in Europe. But it has a major impact on the languages that followed with the introduction of 360 line of computers in 1963, IBM developed NPL (New programming language) at its Jursley Laboratory in England. The name NPL was later changed to MPPL (Multipurpose Programming language) and then to just PL/I. PL/I merged the numerical attributes of FORTRAN with the business features of COBOL.

- **Business Languages :-** After numerical calculations, the next application domain was Business data processing. In 1955, Grace Hopper led a group at UNIVAC to develop FLOWMATIC. The goal was to develop business applications using a form of English-like text. In 1959, U.S. Department of Defense sponsored a meeting to develop CBL (Common Business Language) that used English as much as possible for business applications. The specifications published in 1960, were the designs of COBOL (Common Business Oriented Language). COBOL was revised in 1961 and 1962 standardized in 1968, and revised again in 1974 and 1984.
- **AI Languages :-** The Artificial intelligence (AI) languages began in 1950s with IPL (Information processing language) by the RAND corporation. The major breakthrough occurred when John Maccarthy of MIT designed LISP (List Processing for the IBM 704. LISP 1.5 became the **"standard"**. LISP implementation for many years LISP was designed as a list-processing functional language. The problem domain for LISP involved searching. Automatic machine translation, where strings of symbols could be replaced by other strings, was the natural application domain. COMIT by Yngve of MIT, was an early language in this domain. Since Yngve kept his code proprietary, a group at AT & T Bell **labs** developed their own language SNOBOL. While LISP was a general purpose list processing functional language, PROLOG was a special purpose language whose basic control structure and implementation strategy was based on concepts from mathematical logic.
- **Systems Languages :** The use of assembly language held on for many years in the systems area long after other application domains started to use higher level languages. The reason was need for efficiency. Many systems programming languages, such as CPL and BCPL, were designed, but were never widely used. With the development of C and a competitive environment in UNIX written mostly in C during the early 1970s, high-level languages have been shown to be effective in this environment, as well as in others.

1.4 CHARACTERISTICS OF A GOOD PROGRAMMING LANGUAGE

A programming language provides both a conceptual framework for thinking about algorithms and a means of expressing those algorithms. The languages should be an aid to the programmer long before the actual coding stage. A good programming language should include the following characteristics.

1. **Clarity, Simplicity and Unity :-** The attributes covered under this heading includes.

(*i*) It should provide a clear, simple and unified set of concepts that can be used as primitives in developing algorithms.

(*ii*) The vocabulary of the language should resemble English (or some other human language). Symbols, abbreviations and jargon should be avoided unless they are already familiar to most people.

(*iii*) Programs should consist mostly of instructions ; tedious declarations should be kept to a minimum.

(*iv*) The language and its class or function library should be fully documented.

(*v*) The language should provide full facilities for handling a graphical user interface.

(*vi*) Any concept that cannot easily be explained should not be included in the language.

(*vii*) A program should have minimum number of different concepts, with the rules for their combination being as simple and regular as possible. This attribute is known as **conceptual integrity**.

(*ix*) The syntax of the language should be easy and simple, so that the language is readable and writable.

(x) A good programming language should mirror the semantic differences in the language syntax.

(xi) A good language is designed for the designer to code in.

(xii) A good language is never created with corporate interests in mind.

(xiii) A good language respects the programmer's application domain.

(xiv) A good language gives good expensive power, excellent libraries and run-time support and great documentation.

2. **Orthogonality :-** The term orthogonality refers to the attribute of being able to combine various features of a language in all possible combinations, with every combination being meaningful. Consider for example, a language provides for an expression that can produce a value, and it also provides. For a conditional statement that evaluates an expression to get a true or false value. These two features of the language expression and conditional statement are orthogonal if any expression can be used with in the conditional statement.

When the features of a language are orthogonal, then the language is easier to learn and programs are easier to write because there are fewer exceptions and special cases to remember.

3. **Naturalness for the application :-** This includes the following points :

(i) The syntax of the language should allow the program structure to reflect the underlying logical structure of the algorithm.

(ii) The language should provide appropriate data structures, operations, control structures, and a natural syntax for the problem to be solved.

(iii) It should be possible to translate a program design directly into appropriate program statements that reflect the structure of the algorithm.

(iv) Sequential algorithms, concurrent algorithms, logic algorithms, all have different natural structures represented by programs.

4. **Support for Abstraction :-** This includes the following points :

(i) A good language should allow data structures, data types and operations to be defined and maintained as use if contained abstractions. The programmer may use them in other parts of the program knowing only their abstract properties without concern for the details of implementation.

(ii) The language should probably be object-oriented.

(iii) The degree of abstraction allowed by a programming language and the naturalness of its expensive are very important to writability.

(iv) Programming languages can support two distinct categories of abstraction, **process** and **data**.

5. **Readability :-** An important criteria for judging a programming language is the ease with which the programs can be read and understood. Therefore, a good programming language must consider readability in the context of the problem domain. *e.g.* if a program that describes a computation was written in a language not designed for such use, the program may be unnatural and convoluted, making it unusually difficult to read.

6. **Writability :-** Writability is a measure of how easily a language can be used to create program for a chosen problem domain. Most of the language characteristics that affect readability also affect writability. Therefore, a good programming language must consider writability in the context of the target problem domain of a language.

7. **Portability :-** A language is portable if its programs can be compiled and run on different machines without the source code having to be re-written. This concept of portability or transportability is one of the important criterion for many programming projects. Ada, FORTRAN, C and Pascal all have standardized definitions allowing for portable applications to be implemented.

8. **Generality :-** Generality is related to orthogonality. It refers to the existence of only necessary language features, with others composed in a free and uniform manner without limitation and with predictable effects.

9. **Ease of program verification :-** The reliability of programs written in a language is always a central concern. A program is said to be reliable if it performs to its specifications under all conditions. When an error occurs, it should be easily detected and corrected. For the easy program verification, a good programming language should have following capabilities.

(i) A good programming language should try to find as many defects at compile-time as possible. They are usually statically typed, and enforce type-safety in varying degrees.

(ii) A good programming language should provide type checking which involves testing for type errors in a given program, either by the compiler or during program execution. Type checking is an important factor in language reliability.

(iii) A reliable language should be able to handle run-time errors, unusual conditions detected by the program, take corrective measures and then continue is a great aid to reliability. This language facility is called **exception handling**. Ada, C++ and Java include extensive capabilities for exception handling.

(iv) Some programming language use aliasing (*i.e.*, having two or more distinct referencing methods, or names for the same memory cell) to overcome deficiencies in the languages data abstraction facilities. Other languages greatly restrict aliasing to increase their reliability.

(v) Both readability and writability influence reliability. Programs that are difficult to read are difficult both to write and to modify.

10. Programming Environment :- The presence of an appropriate programming environment may make a technically weak language easier to work with then a stronger language that has little external support. A long list of factors might be included such as reliable and well documented implementation, special editors and testing packages. Facilities for maintaining and modifying multiple versions of a program may make working with large programs much simpler.

11. Fast Translation and Efficient Object Code :-

(i) A good programming language should have fast translators capable of translating source program quickly into object code.

(ii) The object code should be efficient and optimized.

12. Cost of Use :- The ultimate total cost of a programming language is a function of many of its characteristics. Cost is certainly a major element in the evaluation of any programming language.

(i) **Cost of program execution :-** The cost of program execution is of primary importance for large production programs that will be executed repeatedly. The cost of executing programs written in a language is greatly influenced by that language's design.

(ii) **Cost of program creation, testing and use :-** The cost of writing programs in the language is a function of writability of the language, which depends on its closeness in purpose to the particular application. Another aspects of cost in this category includes the cost of implementation, testing and modifying. A language whose implementation system is either expensive or runs only on expensive hardware will have a much smaller chance of ever becoming widely used.

(iii) **Cost of program translation :-** A good criteria for selecting a programming language is the cost of program translation. For example, the student programs are compiled many times while being debugged but are executed only a few times, In such a cases it is important to have a fast and efficient compiler rather than a compiler that produces optimized executable code.

(iv) **Cost of training programmers :-** The cost of training programmers to use the language is a function of the simplicity and orthogonality of the language and the experience of the programmers. Though more powerful languages need not be harder to learn, they often are.

(v) **Cost of Program maintenance :-** The final consideration is the cost of maintaining programs, which includes both corrections and modifications to add new capabilities. The cost of software maintenance depends on a number of language characteristics but primarily readability. Because maintenance is often done by individuals other than the original author of the software, poor readability can make the task extremely challenging.

1.5 PROGRAMMING LANGUAGE TRANLATORS

Over the years, the use of higher level languages for programming has become widespread. However, computers do not understand high level languages. Thus, a programmer's view of the execution [Figure 1.2] cannot be realized in practice.

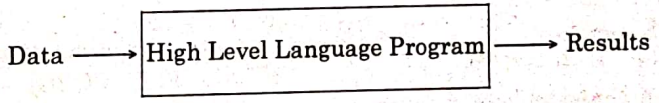


Fig. 1.2 Programmer's View of Program Execution.

In order to execute a high level language program written by a programmer, it is necessary to convert it into the language understood by the machine (computer) – the so called **machine language**. A **translator** is a program which performs translation from a high level language program into the machine languages of a computer.

Therefore, a **translator** is a program that takes as input a program written in one programming language (the source language) and produces as output a program in another language (the object or target language). The schematic for translation and execution of the high level language program is as shown in figure 1.3.

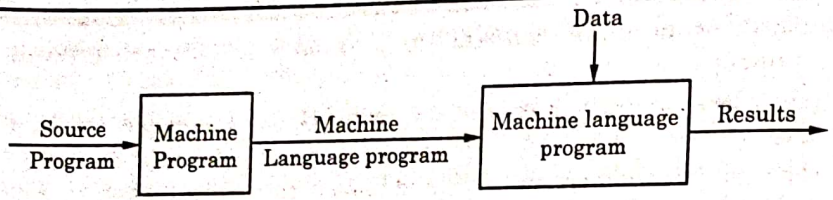


Fig. 1.3 : The Schematic for Translation and Execution of the High Level language Program.

The schematic for programming language translation includes following parts.

(*i*) **Source program :-** The program written in a high-level programming language.

(*ii*) **Object Program :-** The source program after it has been translated into machine language.

(*iii*) **Translator Program :-** The program that translates the source program into object program.

The important tasks of a translator are :

(*a*) Translating source program into object program.

(*b*) Providing diagnostic messages wherever specifications of the source language are violated by the programmer.

(*c*) To create an executable program.

It is not always necessary that the translator's output should be a machine language program. Therefore, depending upon source and object program there are different types of translators. These are discussed as given below.

Compiler :- A compiler is a computer program (or set of programs) that reads a program written in a high level language (*e.g.* FORTRAN, PL/I, COBOL etc) and translates it into an equivalent program in a low level language such as machine language. As an important part of this translation process, the compiler reports to its user the presence of errors in the source program. The concept is illustrated in figure 1.4.

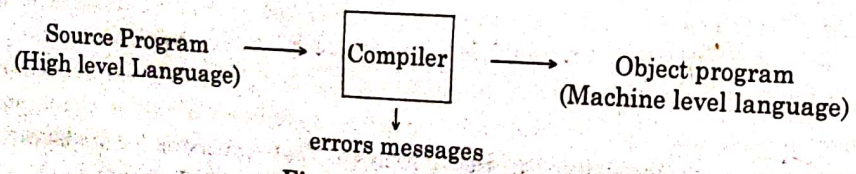


Fig. 1.4. A Compiler.

Executing a program written in a high-level programming language is basically a two-step process :-

(*i*) The source program must first be compiled (*i.e.* translated in object program).

(*ii*) The resulting object program is loaded into memory and executed.

The above two-step process is shown in figure 1.5.

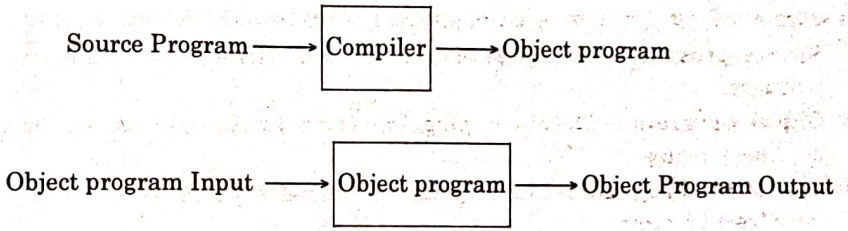


Fig. 1.5. Compilation and Execution.

Advantages :-

1. A compiler translates a program in a single run.
2. It consumes less time.
3. CPU utilization is more.
4. Both syntactic and semantic errors can be checked at the same time.
5. It is easily supported by many high level languages like Pascal, C, C++, etc.

Disadvantages :-

1. It is not flexible.
2. It consumes more space.
3. Error localization is difficult.
4. Even a small error present in a program can cause the whole program to be recognized.
5. An object program produced is usually much larger than the source program that produced it.

Interpreter :- An interpreter is also a program that translator a high-level language program into a low language program, but it does it at the moment the program is run. The interpreter takes the program one line at a time and translates

each line before running it. It translates the first line and runs it, the translates the second line and runs it etc. The interpreter has no "**memory**" for the translated lines, so if it comes across lines of the program within a loop, it must translate them afresh every time that particular line runs. An interpreter, therefore, translates high-level instructions into an intermediate form, which it then executes. The process of interpretation is shown in figure 1.6.

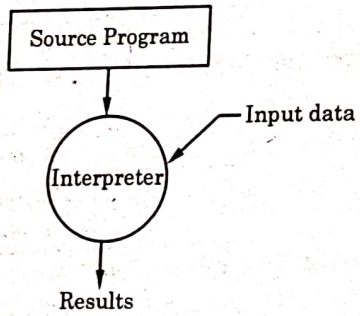


Fig. 1.6. An Interpreter.

Advantages :-

1. An Interpreter translates the program line by line.
2. Interpreters are often smaller in size.
3. Interpreters facilitates the implementation of computer programming language constructs.
4. It is flexible.
5. Error localization is easier.
6. Most command languages such as JCL, in which one communicates directly with the operating system, are interpreted with no prior translation at all.
7. It is easily supported by the languages such as Java.

Disadvantages :-

1. It consumes more time. *i.e.* it is slower.
2. Only syntactic errors are checked.
3. CPU utilization is less.
4. An object program produced is usually much larger than the source program that produced it.
5. Interpreters are less efficient.

The comparison between compiler and Interpreter is as given below in table 1.2.

Compiler	Interpreter
1. A compiler translates the complete source program in a single run.	1. An Interpreter translates the complete source program line by line.
2. It consumes less time *i.e.* it is faster than interpreter.	2. It consumes much more time then compiler *i.e.*. it is slower than compiler.
3. It is more efficient.	3. It is less efficient.
4. CPU utilization is more.	4. CPU utilization is less as compared to the compiler.
5. Both Syntactic and semantic errors can be checked at the same time.	5. Only syntactic errors are checked.
6. Compiler are larger in size.	6. Interpreters are often smaller than compilers.
7. It is not flexible.	7. It is flexible.
8. The localization of errors is difficult.	8. The localization of error is easier than compiler.
9. A presence of an error may cause whole program to be re-organized.	9. A presence of an error cause only a part of the program to be re-organized.
10. Compiler is used by languages such as C, C++.	10. Interpreter is used by languages such as Java.

Table 1.2 : Differences between Compiler and Interpreter

KEY POINTS TO REMEMBER

- The term **'programming language'** is made up of two different words namely **'Programming'** and **'Language'**.
- The **programming** is defined as the art of writing instructions for a computer to solve a specification task.

- A **language** is defined as the set of all possible strings, words sentences that can be derived from a given alphabet (Σ).
- A **programming language** is a notation for writing computer progra used for specifying, organizing and reasoning about computations. T three basic components of a programming language are syntax, semant and pragmatics.
- The main reasons to study different kinds of programming languages ar
  - To improve your ability to develop effective algorithms
  - To improve your use of existing programming language
  - To increase your vocabulary of useful programming construct
  - To allow a better choice of programming language
  - To make it easier to learn a new language
  - To make it easier to design a new language
  - Overall advancement of computing
- The language developments from mid-1950s to the early 1975s can summarized as :
  - Numerically based language
  - Business languages
  - AI languages
  - System languages
- The main characteristics of a good programming language are :
  - Clarity, simplicity and unity
  - Orthogonality
  - Naturalness for the application
  - Readability.
  - Writability
  - Portability
  - Generality
  - Ease of program verification
  - Programming environment
  - Fast translation and efficient object code.
- A **translator** is a program which performs translation (a conversion) f a high level language program into the machine language of a compute

- The schematic for programming language translation include source program, object program and translator program.
- The two main type of translator are compiler and interpreter.
- A compiler translates the complete source program in a single run while an interpreter translates the source program line by line.
- Compiler is used by the languages like C, C++ etc while interpreter is used by the languages such as Java.

EXERICISE

1. Define the term programming language ? How it is different from natural language ?
2. What are the main reasons to study different kinds of programming languages ?
3. Write a short note on history of programming language.
4. What are the characteristics of good programming language ?
5. Define the term translator. What are different types of translation ?
6. Differentiate between compiler and interpreter.
7. Write short on :
   (*a*) Numerically based languages
   (*b*) Business languages
   (*c*) AI languages
   (*d*) System languages.

CHAPTER

2 ELEMENTARY DATA TYPES

2.1 INTRODUCTION

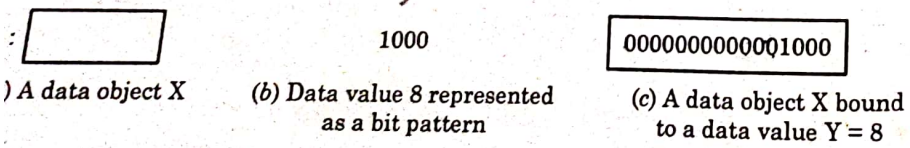
Data, operations, and control are the basis of discussion and comparison of programming languages. Languages provide the programmer with certain basic data types, specifying both the set of data items and a set of operations on them. The computer programs produce results by manipulating data. An important factor in determining the ease with which they can perform this task is how well the data types match the real-word problem space. It is therefore crucial that a language support an appropriate variety of data types and structures. The various data types available in any programming language are classified broadly as :

1. Elementary data types
2. Structured data types

The elementary data types are the part of this chapter while the structured data types are discussed in chapter 4.

2.2 DATA OBJECTS, VARIABLES AND CONSTANTS

The term data object refers to something meaningful to an application. A data object represents a place where data values may be stored and later retrieved. A data value might be a single number, character or possibly a pointer to another data object. A data value is represented as a bit pattern. If we say that a data object contains a data value Y, it means that a storage block that represents X contains a particular bit pattern representing Y as shown in figure 2.1.



) A data object X *(b) Data value 8 represented as a bit pattern* *(c) A data object X bound to a data value Y = 8*

***Fig. 2.1.** A Data Object X with a Data Value Y.*

A data object is said to be

1. **Elementary**, if the data value it contains can be manipulated as a unit *e.g.* an integer data object is an example of elementary data object.
2. **Structured**, if a data object is composed of other data objects *e.g.* an array data object may be composed of number of similar data objects like integers.

The term data objects also refers to run time grouping of one or more piece of data in a virtual computer. The data objects that exist during the program execution can be

1. **System Defined :** These are the data objects set by the system itself and the programmer cannot access these data objects directly *e.g.* subprogram activation records, file buffers, run time storage stacks, free space lists etc.
2. **Programmer Defined :** These are the data objects created explicitly by programmers through declarations and statements in the program *e.g.* variables, constants, arrays, files etc.

In the last we summarize the concept of data objects by following remarks :

1. A data object is represented as storage while a data value is represented as a pattern of bits.
2. Some data objects exist at the start of program execution and others are created dynamically during the execution.
3. The most important attributes of data objects are type, location, value, name and component.

2.2.1 NAMES

A **name** is a string of characters used to identify some entity (data object) in a program. The earliest programming languages used single – character names. Most of the languages today use multi-character names. Some languages, such as C + +, do not specify a length limit on names, although implementers of those languages sometimes do.

In a program, names may be assigned to data objects such as procedures, labels, types and more. Many languages use certain names as part of their syntax (*e.g* for, while etc) or as special functions or operators (mod, read, printf). Any word whose meaning is predefined and cannot be redefined by the programmer is called a **reserved word**. There are often a number of words which are not reserved but have a predefined meaning. These are called **keywords** and can be defined by the user for another meaning. For example in Pascal the predefined types like integer real and predefined functions like trunc, sqrt are not reserved.

2.2.2 VARIABLES

A program **variable** is an abstraction of a computer memory cell or collection cells. Programmers often think of variables as names for memory location Formally, a variable is defined as a data object that is defined and named by programmer explicitly in a program.

A variable can be characterized by a set of six types describing the attributes variables. These attributes are : name, address, value, type, lifetime and scop Some programming languages like Pascal do not distinguish between variab names written in upper case and lower case letters while some other languages li C makes distinction between upper case and lower case variable names. *e.g* sala and SALARY are treated as same variables in Pascal while in C these are consider as two distinct data objects. Consider the following declaration in C language

```
int a;
```

Here *a* is a simple variable of type integer.

2.2.3 CONSTANTS

A **constant** is defined as a data object that is bound to a value permanent during its lifetime. For example consider the following declarations.

```
const int SUM = 16 ;
```

This declaration species that the data object named SUM is bound permanent to value 16 and is therefore treated as a constant. The advantages of using constan in a program are :

1. The use of constants in a program improves the readability and progra reliability *e.g.* using the name pi instead of the constant 3.14159.
2. The programs processing a fixed number of data values, say 100, general use the constant 100 in a number of locations for declaring array subscri ranges, for loop control limits, and other uses. A modification to progra deals with finding and changing all occurrences of constant 100 which c be tedious and error prone. By declaring 100 as a named constant requir changing only one line, regardless of the number of times it is used in t program.

A constant used in a program can be

(*i*) **Literal constant** : A constant whose name is just the writt representation of its value *e.g.* "16" is the written decimal representati of the literal constant that is a data object with value 16.

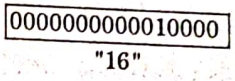


Fig 2.2 : A Literal Constant.

(*ii*) **Programmer – defined or Manifest Constant :** A constant whose name is chosen by the programmer in the definition of that data object. For example, in the declaration

```
const int SUM = 16 ;
```

the constant SUM is a programmer defined constant because the programmer explicitly defines the name SUM for the value 16 as shown in figure 2.3.

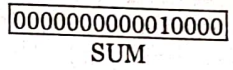


Fig. 2.3 A programmer Defined or Manifest Constant.

2.3 DATA TYPES

The data types of a language are a large part of what determines the language's style and use along with control structures, they form the heart of a language.

A data type is defined as the class of data objects together with a set of operations for creating and manipulating them. The data types that a programming language provides can be broadly classified into following two categories :

1. **Primitive data types :** The data types that are built into the language and are not defined in terms of other types are called **primitive** data types. Many languages include such primitive types as integer, real, character, Boolean and pointer. Some of the primitive types are merely reflections of the hardware, for example, integer types. Other require only a litter non-hardware support for their implementation.
2. **Non-Primitive or Use-defined Data Types :** Many programming languages permit the programmer to define new data types. These user-defined data types are also called **non-primitive data types**. The user-defined data types includes enumeration and subrange types that add to the readability and reliability of programs.

The data types may also be classified as :

1. Elementary data types,
2. Structured data types

These are two types are discussed in detail in next section.

The two concepts regarding data types are

1. **Specification of Data Type :** The basic elements of a specification of a data type are :

(a) The **attributes** that distinguish data objects of that type *e.g* in the specification of array data type the attribute might be number of dimensions, data type of each element etc.

(b) The **values** that data objects of that type may have *e.g* in the specification of array data type, the values would be the set of numbers that form valid values for arrays.

(c) The **operations** that define the possible manipulations of data objects of that type *e.g* in the specification of array data type can be subscripting to select individual array element, to perform arithmetic operations on pair of arrays etc.

2. **Implementation of Data Type :** The basic elements of an implementation of a data type are

(a) The **storage representation,** used to represent data objects of the given data type in this storage of computer during program execution *e.g.* for character values, the hardware or operating system character codes are used.

(b) **Implementation of Operations,** defined for data objects of the given data type in terms of algorithms or procedures that manipulate the chosen storage representation of the data objects.

2.4 ELEMENTARY DATA TYPES

As described in section 2.2, a data object is called an **elementary data object** *if the value it contains can be manipulated as a single unit. Such a class of elementary data objects with a set of operations for creating and manipulating them is termed as an* **elementary data type**. *The examples of elementary data types include integer, real, character, boolean, pointer etc.*

4.1 SPECIFICATION OF ELEMENTARY DATA TYPES

As described earlier the basic elements of specification are

(a) Attributes

(b) Values, and

(c) Operations

These are described with reference to elementary data types as.

(a) **Attributes :** The term attribute in general refers to characteristics or group of characteristics that distinguish one data object from others. The main attributes of a data object are its name, associated address and data type. The following declaration in C.

```
int a ;
```

It specifies that a data object named '**a**' is of type integer. The attributes of a data object may be stored in a collection of memory cells, called **descriptor** (or **dope vector**). A descriptor is the collection of the attributes of a variable. If the attributes are all static, descriptors are required only at compile time. They are built by the compiler, usually as a part of the symbol table, and are used during compilation. For dynamic attributes, however, part or all of the descriptor must be maintained during execution. In this case, the descriptor is used by the run-time system. In all cases, descriptors are used for type checking and by allocation and deallocation operations. The attributes are generally invariant during the lifetime of data object.

(b) **Values :** It refers to set of all possible values that a data object may contain. The values that a data object may assume are determined by the type of that data object. An elementary data object contain a single value from the set of values at any point during its lifetime. For example, the C declaration

```
int a ;
```

specifies that the data object 'a' can assume a single integer value from a set of integer values. However, the value contained in a data object may change (by an assignment statement) during the life time of the data object and is therefore represented explicitly during the program execution.

(c) **Operations :** An operation in general refers to a mathematical function for the manipulation of data objects. An operation includes :

(*i*) **Domain :** It refers to set of all possible input arguments on which the operation is defined.

(*ii*) **Range :** It refers to set of all possible results that an operations may produce as an output.

(*iii*) **Action :** The action of the operation defines the results produced for any given set of arguments.

(*iv*) **Algorithm :** It refers how to compute the results for any given set of arguments. It is used for specifying the action of an operation.

(v) **Signature :** A signature of an operation specifies the number, order and data types of the arguments in the domain of an operation as well as the order and data type of the resulting range. The signature of an operation are specified using the notation.

op_name : arg1 type × arg 2 type × ···× arg n type → result _type

For example, the specification of integer addition is given as

+ : integer × integer → integer

meaning that an addition operation when provided with two integer data objects produces an integer data objects as an output.

Implementation of elementary data types :

As described earlier, the basic elements of implementation are

(a) The storage representation and

(b) The Implementation of operations

These are described with reference to elementary data types as :

(a) **The storage representation :** The representation of a data object is ordinarily independent of its location in memory. Thus, each data object of a given type has the same representation regardless of its particular position in the computer memory. The storage representation is usually described in terms of the size of the block of memory required and the layout of the attribute and data values with in this block. Usually the address of the first word or byte of such a block of memory is taken to represent the location of the data object.

The storage for elementary data types is strongly influenced by the underlying computer that will execute the program. If the hardware storage representations are used, then the basic operations on data of that type may be implemented using the hardware provided operations. If the hardware storage representations are not used then the operations must be software simulated and the same operations will execute less efficiently.

(b) **The Implementation of Operations :** The operations defined for the data objects of a given type may be implemented in one of three main ways :

(i) **Directly as a hardware operation,** *e.g.* the addition and subtraction operations may be implemented using the arithmetic operations built into the hardware if the integer are stored using the hardware representation for integers.

(ii) **As a function subprogram or procedure,** *e.g.* the a square root operation might be implemented as a square-root subprogram that calculates the square-root of its argument.

(iii) **As an inline code sequence :** An in-line code sequence is a software implementation of the operation in which the operations in the subprogram are copied into the program at the point where the subprogram would otherwise have been invoked *e.g.* the absolute-value function.

abs (x) = if $x < 0$ then $-x$ else x

can be implemented as an in-line code sequence as

1. Fetch the value of x from memory.
2. If $x > 0$; skip the next instruction.
3. Set $x = -x$
4. Store the new value of x in memory.

where each line is implemented by a single hardware operation.

5 DECLARATIONS

Before a data object can be referenced in a program, it must be bound (or sociated) to a name and a data type A declaration in a program refers to a tement that provides the information about the name and type of data objects to programming language translator. For example, consider the following C claration :

int *a*, *b* ;

This declaration provides the programming language translator with the rmation that ***a*** and ***b*** are the data objects of type integer are needed during the cution of the subprogram. The declaration also specifies the binding of the data cts to the name ***a*** and ***b*** during their lifetimes.

1 PURPOSES FOR DECLARATION

The various purposes for declarations are as given below :

1. **Type Checking :** The declarations allow the programmer for static type checking *i.e.* checking the types of data objects at compile time rather than at execution time.
2. **Choice of Storage Representation :** As we have seen that the declaration provides the information about type of the declared data object which helps the programming language translator to determine the best

possible storage representation for that data object. This helps in reducing the overall storage requirement and execution time for the program being translated.

3. **Storage Management :** The declarations also serve to indicate the desired lifetime of the data object that makes it possible to use more efficient storage management procedures during program execution. For example in C some of data objects may be declared at the beginning of a subprogram while some other data objects are created dynamically by the use of a special function **malloc**.

4. **Polymorphic Operations :** An operation is said to be polymorphic operation if an operation may take on a variety of implementations depending upon the types of its arguments. The declarations allow the programming language translator to determine at compile time the particular operation designated by an overloaded operation symbol. For example in C, the declaration of data objects *a* and *b* helps in determining the possible addition operation (integer addition or float addition) designated by $a + b$.

2.5.2 EXPLICIT AND IMPLICIT DECLARATIONS

There are basically two types of declarations as discussed below.

1. **Explicit declaration :** An explicit declaration is a statement in a program that lists data object names and specifies that they are a particular type. The explicit declaration create static binding to type. The example of explicit declaration consist of following C declaration.

```
int  a, b ;
```

2. **Implicit declaration :** An implicit declaration (or default declaration) is declaration that hold when no explicit declaration is given. For example, FORTRAN, an identifier that appears in a program that is not explicitly declared is implicitly declared according to the following to convention : the identifier begins with one of the letters I, J, K, L, M or N it is implicitly declared to be of INTEGER type ; otherwise it is implicitly declared to be REAL type. For example, INDEX is assumed to be of integer type with an explicit declaration, otherwise it is of REAL type.

2.5.3 *DEFINITIONS AND DECLARATIONS*

Some programming languages like C and C++, distinguish between declarations and definitions. The declarations specify types and other attributes but do not ca

allocation of storage. Definitions specify attributes and cause storage allocation. The differences between definitions and declarations are given below in table 2.1.

	Definition		Declaration
1.	The definition specify attributes and cause storage allocation.	1.	The declaration specify types and other attributes but do not cause allocation of storage.
2.	The information contained in the definition is only used during translation. The language translator enters the information from type definition into table during translation and whenever the type name is referenced in a subsequent declaration, used the tabled information to produce the appropriate executable code for setting up and manipulating the desired data objects during execution.	2.	The information contained in the declaration of variable is used primarily during translation to determine the storage representation for the data object and for storage management and type checking purposes.
3.	For a specific name, a C program can have only a single definition.	3.	For a specific name, a C program can have any number of compatible declarations.
4.	The type definition allows some aspects of translation such as determining storage representations to be done only once for a single type definition.	4.	The type declaration determines storage representation many times for different declarations.
5.	Inclusion of type definitions in a language does not ordinarily change the run-time organization of the language implementation.	5.	Inclusion of type declaration in a language may change the run-time organization of the language because they are used to set up the run time data objects.

Table 2.1 : Differences between a definition and a declaration

2.6 TYPE CHECKING AND TYPE CONVERSION

The data storage representations built into computer hardware stores the data in the form of binary bit sequences which do not provide any idea regarding the data type of a particular data object. Type checking is the activity of ensuring that the operands of an operator are of compatible types. A compatible type is one that is either legal for the operator or is allowed under language rules to be implicitly converted by compiler-generated code to a legal type. This automatic conversion is called a **coercion**. A type error is the application of an operator to an operand of an inappropriate type. In order to illustrate the concept of type checking consider the following statement.

$$c := a + 3 * b ;$$

Here b must be of a type that permits multiplication by an integer. Similarly, the operands for addition and assignment can be evaluated.

The example given above illustrates the fact that type checking involves evaluating expressions for type compatibility *i.e.* checking that each operation executed by a program receives the proper number of arguments of the proper data type. The type checking may be done at **compile time (static type checking)** or at **run time (dynamic type checking)**. These are described as given below.

1. **Static type checking :** If all bindings of variables to types are static in a language, then type checking can nearly always be done statically. It is much better to detect errors at compile time that at run time because the earlier correction is usually less costly. Therefore, the static type checking involves examining the program text, usually during translation (or compilation). Using the rules of a type system, a compiler can infer from the source text that a function f will be applied to an operand a of the right type, each time the expression $f(a)$ is evaluated. The needed information for static type checking is usually provided in part by declarations that the programmer provides and in part by other language structures.

Advantages : The static type checking involves following advantages :

1. The early detection of errors at compile time.
2. Checking all execution paths.
3. No requirement of type tags on data objects at run time.
4. Substantial gain in efficiency of storage use.
5. Substantial gain in execution speed.

)isadvantages : The static type checking involves following disadvantages :

1. Reduced programmer flexibility.
2. Some languages like APL and SNOBOL4, because of their dynamic type binding, allow only dynamic type checking.
3. In languages without declarations, no static type checking is possible.
4. Static type checking tends to affect declarations, data control structures, and provisions for separate compilation of subprograms.
5. Static type checking is complicated when a language allows a memory cell to store values of different types at different times during execution.

2. **Dynamic type checking :** If all bindings of variables to types are ynamic in a language, then type checking can nearly always be done dynamically. he dynamic type checking is done during program execution. The dynamic type ecking is usually implemented by storing a type tag in each data object that dicates the data type of the object. For example, an integer data object would ntain both the integer value and an **"integer"** type tag. Each operation is then plemented to begin with a type-checking sequence in which the type tag of each gument is checked. The operation is performed only if the argument types are rrect ; otherwise an error is signaled. The operation must also attach the ppropriate type tags to its results that subsequent operations can check them *i.e.* ynamic checking is done by inserting extra code into the program to detect npending errors.

dvantages : The dynamic type checking involves following advantages :

1. Flexibility in program design.
2. No requirement for declarations.
3. The type of a data object associated with a variable name may be changed as per need during the program execution.
4. The dynamic type checking can detect many errors that can not be detected by static type checking.
5. In most languages, static type checking is not possible for some language constructs in certain cases but the same purpose may be achieved by dynamic type checking.

isadvantages : The dynamic type checking involves following disadvantages :

1. The dynamic type checking takes up more space as the dynamic type checking involves inserting extra code into the program to detect impending errors.

2. Programs are difficult to debug as all possible execution paths are explored during program testing.
3. The dynamic type checking takes more time which reduces the speed executing the operation.
4. Less efficient than static type checking.
5. More expensive.
6. Errors can lurk in a program until they are reached during execution.
7. Large programs tends to have portions that are rarely executed, so program could be in use for a long time before dynamic checking detect type error.
8. Properties that depend on values computed at run time are rarely check For example, imperative languages rarely check that an array index is w in bounds.
9. The underlying hardware do not provide support for dynamic type checki
10. Managing the tags increases the complexity.

2.6.1 STRONG TYPING

One of the new ideas in language design that became prominent in the so cal structured programming revolution of the 1970s is **strong typing**. The term **stro** and **week** refers to the effectiveness with which a type system prevents errors type system is **strong** if it accepts only safe expressions. In other words, expressi that are accepted by a strong type system are guaranteed to evaluate without a t error. A type system is **weak** if it is not strong.

Consider the figure 2.4 where a strong type system accepts some subsets of s programs ; however we have no information about the size of this subset.

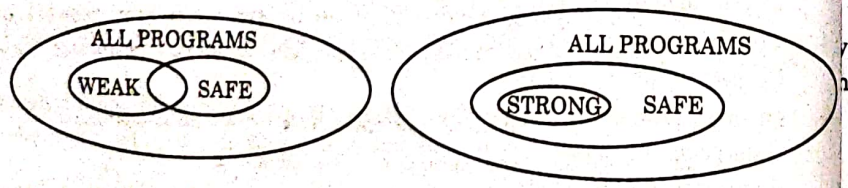


Fig. 2.4 A Weak and Strong Type System.

In more general words, a strongly typed language is one in which each name a program in the language that a single type associated with it and is known compile time. The essence of this definition is that all types are statistically bou

The weakness of this definition is that it ignores the possibility that, although a variables type may be known, the storage location to which it is bound may store values of different types at different times. To take this possibility into account, we define a programming language to be strongly typed if type errors are always detected. This requires that type rules are strictly enforced at both compile and run time. If type rules are not enforced, despite implicit or explicit type declarations, the language is considered weakly typed.

Another useful definition of strong typing is given as :

1. Every object in the language belongs to exactly one type.
2. Type conversion occur by converting a value from one type to another, conversion does not occur by viewing the representation of a value as a different type.

Examples :

1. **Pascal** is nearly strongly typed, but it fails in its design of variant records because it allows emission of the tag that stores the current type of a variable.
2. **Ada** is nearly strongly typed. References to variables in variant records are dynamically checked for current type values. However, Ada allows programmers to breach the Ada type-checking rules by specifically requesting that type checking be suspended for a particular type conversion.
3. **C** and **C++** are not strongly typed languages because both allow functions for which parameters are not type checked. Further more, the union types of these languages are not type checked.

.6.2 TYPE CONVERSION

Although true strong typing is difficult, if we restrict conversion between one ype and another, we come close to strong typing. The mismatch occurring during he type checking may be

1. Either flagged as an error, and an appropriate error action taken, or
2. A coercion (or implicit type conversion) may be applied.

The type conversion is an operation which takes a data object of one type and roduces the corresponding data object of a different type. The signature of a type nversion operation is given as

`conversion_op : type1 → type2`

Type conversions are either **narrowing** or **widening (promotion).**

(a) **A narrowing conversion** converts a value to a type that cannot store e approximation of all of the values of the original type. For exam converting a real to an integer or converting a double to a float in C. An range of source data type (the data type required to be converted) is m larger than that of target data type (the resultant data type obtai therefore the information gets lost.

(b) **A widening conversion** converts a value to a type that can include least approximations of all of the values of the original type, For exam converting an int to a float in C. Another example includes convertin short int to long int in C. As the range of target data type is much la than that of source data type, therefore there is no loss of information.

Type conversions can be either **explicit** or **implicit (coercion)**. These discussed as given below :

1. Implicit type conversion (Coercions) : The programming langu that allow mixed-mode expressions must define conventions for implicit oper type conversions. A **coercion** is defined as an automatic conversion between ty For example in Pascal, if the operands for the addition operation is of integer and other of real type, one of then the integer data object is implicitly converte type real before the addition is performed. As another example, in C, characters implicitly coerced to integers. The typical use is for character input as show following program.

```
# include <stdio.h>
        main ( )
            {
            int c ;
            c = getchar ( ) ;
            while  (c ! = EOF)
            {
            put char (c) ;
            c = getchar ( ) ;
            }
    }
```

Program 2.1. A sample C Program

The function getchar ( ) returns an integer. Such an integer, either corresponds to a character or it is special integer constant EOF that cannot be confused with any character as shown in figure 2.5.

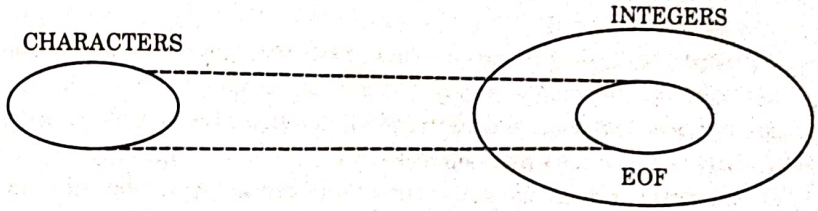


Fig. 2.5. The Coercion of Characters into Integers in C.

The function getchar ( ) returns EOF if there are no more characters to be read – *i.e.*, when the end of the input file is reached. Declaring c to be an integer allows it to take on the integer value of EOF. In fact, an assignment c = getchar ( ) results in a coercion because the character on right side is coerced to an integer when the assignment occurs.

2. Explicit Type Conversion : Most languages provide some capability for doing explicit conversions, both widening and narrowing. In some cases, warning messages are produced when an explicit narrowing conversion results in a significant change to the value of the object being converted. For example, Pascal provides a built-in function **round** that converts a real-number data object to an integer data object with a value equal to the rounded value of the real. In C based languages, explicit type conversions are called **casts.** The desired type is placed in parentheses just before the expression to be converted, as shown in **(int)** X for **float** X converts the value of X to type integer. One of the reasons for the parentheses in C conversions is that C has several two-word type names, such as **long int.**

2.6.3 ADVANTAGES OF TYPE CONVERSION

These are following advantages of the type conversion which are given as below :

1. If during type checking, a mismatch occurs between the actual type of an argument and the expected type for that operation, then type conversion easily converts the data object implicitly and prevents the error.
2. In some languages such as 'C', type conversion is a built in function, which itself, implicitly cast an expression to convert it to the correct type.
3. Type conversion is automatically invoked in certain cases of 'mismatch'. For example, in Pascal, if the arguments for an arithmetic operation such as '+'

are of mixed real and integer type, the integer data object is implicitly converted to type real before the addition is performed.

4. No information is lost. Since every short integer can be represented as a long integer, no information is lost bye automatically invoking a **short int → long int.**
5. With dynamic type checking, conversions or coercions are made at the point that the type mismatch is detected during execution. For such languages narrowing conversions could be allowed. For static type checking, extra code is inserted in the compiled program.
6. Type conversion is subtle need when there are large number of data types in a programming language.

2. ASSIGNMENT AND INITIALIZATION

An assignment statement is one of the central constructs in imperative languages to dynamically change the binding (or association) of values to variables. The general syntax of the simple assignment statement is given as :

<target_variable> <assignment_operator> <expression >

For example, the assignment statement

$$a := b + c ;$$

Here, the assignment statement computes the value of the expression $b + c$ and associates it with a ; the old value of a is forgotten.

In general an assignment statement consist of

(*a*) An assignment operator (= or : =)

(*b*) References to the left of assignment operator called *l*-value, which refers to the location of data object that will contain the new value.

(*c*) References to the Right of assignment operator called *r*-value, which refers the value contained in the named data object.

Based on above three parameters an assignment operation is defined as

1. Compute the *l*-value of the first operand expression.
2. Compute the *r*-value of the second operand expression.
3. Assign the computed *r*-value to the computed *l*-value data object.
4. Return the computed *r*-value as the result of the operation.

Example 2.1 : Consider the following assignment statement in C.

A = B

Show the effect of above assignment statement if

(*a*) Both A and B are integer variables.

(*b*) Both A and B are pointer variables.

Solution. (*a*) When A and B are both integer variables then the given assignment statement means **"Assign a copy of the value of variable B to variable A"** *i.e.* assign to the *l*-value of A the *r*-value of B. Suppose initially A contains 0.9 and B contains 0.5. The effect of assignment is shown in figure 2.6.

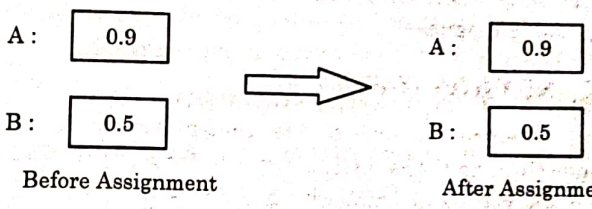


Fig. 2.6. An Assignment Statement when Both A and B are Integer Variables.

(*b*) When A and B are both pointer variables. If B is a pointer, then B's *r*-value is the *l*-value of some other data object. This assignment means **"Make the *r*-value of A to the same data object as the *r*-value of B".** *i.e.* assign to the *l*-value of A the *r*-value of B which is the *l*-value of some other data object. This means that a copy of the pointer stored in variable B is assigned to variable A as shown in figure 2.7.

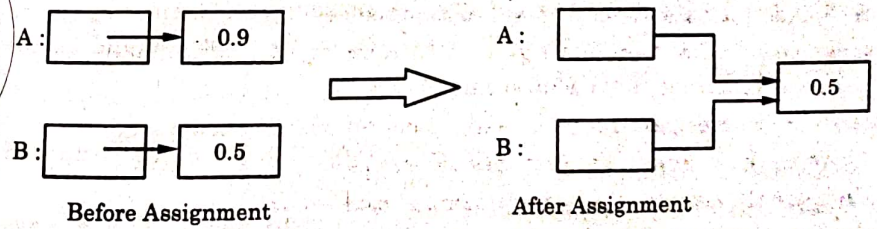


Fig. 2.7. An Assignment Statement when Both A and B are Pointer Variables.

Example 2.2. Write the signature of an assignment statement in C an Pascal.

Solution. (*a*) In C language, the signature of an assignment operation is

= : int 1 × int 2 → int 3

This means that the value contained in data object int 1 to be a copy of the valu contained in data object int 2 and also create and return a new data object int containing a copy of the new value of int 2.

(*b*) In Pascal language, the signature of an assignment operation is

: = : int 1 × int 2 → void

This means the value contained in data object int 1 is set to be a copy of th value contained in data object int 2 and return no explicit result.

2.7.1 SOME VARIANTS OF ASSIGNMENT STATEMENTS

In addition to simple assignment statements as described above, these are som variants of assignment statement as explained below :

1. **Multiple targets :** Some languages like PL/I allow assignment of th expression value to more than one location *e.g.* the statement

   a, b = 0;

   assigns the value zero to both *a* and *b*.

2. **Conditional targets :** Some languages like C ++ and Java allow conditional targets on assignment statements *e.g.* the statement

   flag ? count 1 : count 2 = 0 ;

   means

   if *(flag)* count 1 = 0 ; **else** count 2 = 0 ;

3. ***Compound Assignment* Operators :** A compound assignment operation has the destination variable also appearing as the first operand in the expression on the right side *e.g.*

   a = a + b ;

   The languages like C, C + + and Java have versions of the compound assignment operation for most of their binary operators.

4. **Unary Assignment Operators :** The languages like C, C + +, Java include two special unary arithmetic operators (+ + for increment and – – for decrement) that are actually abbreviated assignments. The operators + + and – – can appear as either prefix operators or postfix operators. For example, the statement

   a = + + *b* ;

is equivalent to

b = *b* + 1 ;
a = *b* ;

If the same operator is used as a postfix operator as in

a = *b* + + ;

This is equivalent to

a = *b* ;
b = *b* + 1 ;

5. **Mixed – Mode Assignment :** The languages like C, C + + and FORTRAN use coercion rules applied to all the operands in the right side to the type of the target before evaluation *e.g.* in C.

```
int a, b ;
float c ;
.....
c = a/b ;
```

Because *c* is float, the value of *a* and *b* could be coerced to float before the division, which could produce a different value for *c* that if the coercion were delayed (for example, if *a* were 2 and *b* were 3).

2.7.2 INITIALIZATION

A data object that has been created but not initialized is a serious source of programming error for professional programmers as well as for beginners. The random bit pattern contained in the value storage area of an uninitialized data object ordinarily cannot be distinguished from a valid value, since the valid value also appears as a bit pattern. Thus a program often may compute the value of an uninitialized variable and appear to operate correctly, when in fact it contains a serious error. Thus a data object needs to be initialized before the code of the program or subprogram in which they are declared begins executing.

The binding of a variable to a value at the time it is bound to storage is called **initialization.** If the variable is statistically bound to storage, binding and initialization occur before run time. If the storage binding is dynamic, initialization is also dynamic. The example of initialization includes :

A : array (1 ... 3) of float : = (19.3, 14.3, 16.3) ;

initialize an array A with three real numbers.

Another example include as in the Ada declaration

SUM : INTEGER : = 0 ;

Neither Pascal nor Module –2 provides a way to initialize variables, except at run time with assignment statements.

In general, initialization occurs only once for static variables, but it occurs with every location for dynamically allocated variables, such as the local variables in an Ada procedure.

2.8 NUMERIC DATA TYPES

Some form of numeric data is found in almost every programming language. The properties of numeric data representations and arithmetic on computers differs substantially from the number and arithmetic operations discussed in ordinary mathematics.

Many early programming languages had only numeric primitive types. These types still play a central role among the types supported by contemporary languages. The various numeric data types available in various programming languages are :

1. Integer
2. Real numbers
3. Subrange
4. Complex numbers
5. Rational numbers

These data types are explained as given below :

1. **Integer :** One of the most common primitive numeric data types is integer. The specification and implementation of integers is described below :

Specification : The specification of integers includes :

(i) **Attributes :** The main attribute of integers is its type.

(ii) **Values :** The values associated with integers are determined by the underlying machine, with the largest and smallest numbers determined primarily by the number of bits in a machine word. The supported integer values are between – Max Int and + MaxInt, where MaxInt is a predefined constant for an implementation. For example, if a machine supports 2s-complement arithmetic with a 16-bit word, using one-bit for the sign the largest 15-bit value would be + 32,767. Hence this would likely become the value of MaxInt on this machine for a language like Pascal. As another example, C has four different integer specifications : int, short, long and char.

(iii) **Operations :** The various operations that can be applied on integers are :

- **Arithmetic Operations :** The normal arithmetic operations are addition, subtraction etc. An arithmetic operations is called **Binary operation** if it requires two operands. The binary operation is specified as

  Bin_op : **int** × **int** → **int**

  e.g. + : **int** × **int** → **int** [Addition of two integers is another integer]
  − : **int** × **int** → **int** [Subtraction of two integers is another integer]
  * : **int** × **int** → **int** [Multiplication of two integers is another integer]

  An arithmetic operation is called **unary** if it involves only one operand. The unary operation is specified as :

  Unary)_op : **int** → **int**

  Uniminus (–) : int → int [n uniminus operation]

  Bit Operations : The bit operations are specified as

  Bin_op : **int** × **int** → **int**

  The examples of bit operations are **and** the bits together (&), or the bits together (|) and shift the bits (< <), among others.

- **Relational Operations :** The relations operations are specified as

  Rel_op : **int** × **int** → **Boolean**

  The examples of Relation operation are equal, not equal, less than, greater than, less than-or equal and greater-then-or-equal.

- **Assignment :** As discussed earlier, the assignment statement may assume any of the following two specifications.

  assignment : int × int → void [*e.g* in Pascal]
  assignment : int × int → int [*e.g* in C]

(b) **Implementation :** The implementation of integers include.

(i) **Storage Representation :** The storage representation for integers is almost always the integer representation for numbers used in the underlying hardware. There are three binary possible storage representations for integers.

- **A representation having no run-time descriptor and only the value in stored.** The representation is useful for languages providing declarations and static type checking *e.g.* in C and FORTRAN. Such a representation is given in Figure 2.8.

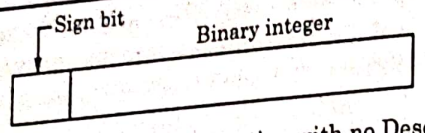


Fig. 2.8. An Integer Representation with no Descriptor.

→ **A representation having a run-time descriptor stored in a separate memory location, with a pointer to the "full-word" integer value.** The main advantage of this representation is that hardware arithmetic operations may be used. The main disadvantages of this representation is the approximately double the storage required for a single integer data object. The representation is mainly used in LISP. Such a representation is given in Figure 2.9.

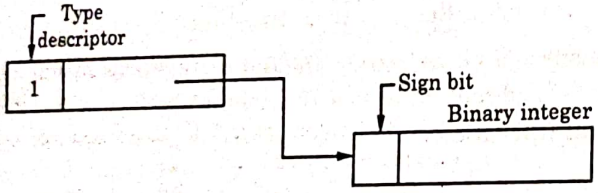


Fig. 2.9. An Integer representation with Descriptor Stored in Separate Word.

→ **A representation that stores the descriptor and value in a single memory location.** The main advantage is the reduced storage while the main disadvantage is that the hardware arithmetic operations cannot be used without first clearing the descriptor from the integer data object, performing the arithmetic, and then reinserting the descriptor. This representation is mainly suitable for hardware– implemented type descriptors and is shown in figure 2.10.

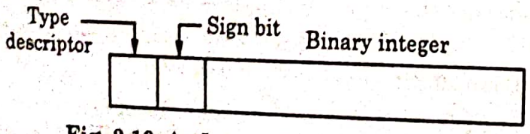


Fig. 2.10. An Integer Representation with no Descriptor Stored in Same Word

Implementation of Operations : If the integers are stored using the hardware representation for integers, then the integer operations (*e.g.* addition and subtraction) may be implemented using the arithmetic operations built into the hardware.

2. Real Numbers : The computer representation of real numbers differs significantly from the concept in a mathematics course, in which most real numbers do not have an exact decimal representation. In computer languages, the actual value may be represented only by an approximation. For example, pi and sqrt (2) have non-repeating and non-terminating decimal representations in math, but must be approximated by some digital value for computer use. The real numbers are classified as

(*a*) Fixed point real numbers (Decimal Data types)

(*b*) Floating point real numbers

These are described as given below.

(a) Fixed – Point real numbers :

Specification : One of most important attribute of fixed point real numbers is its type. The fixed point number representation specifies both a fixed number of digits and the position of the decimal (or binary) point. They are much like integers then, except for the radix (decimal or binary) point. These are primary data types for business data processing and are therefore essential to COBOL. In COBOL, a fixed point variable is declared as

X PICTURE 999V99

which declares X as a fixed-point variable with three digits before the decimal and two digits after.

Implementation : The fixed point real numbers are stored very much like character strings, using binary codes for the decimal digits. These representations are called binary coded decimal (BCD). In some cases, they are stored one digit per byte, but in others they are packed as two digits per byte.

The operations on fixed point numbers are done in hardware on machines that have such capabilities, otherwise, they are simulated in software. For example, in PL/I, fixed data are of type **FIXED DECIMAL**. We can write

DECLARE X FIXED DECIMAL (10, 3) ;

It signifies that X is 10 digits with three decimal places. We store such data as integers, with the decimal point being an attribute of the data object. If X has the value 222.016, the *r*-value of X will be 222,016 and the object X will have an

attribute scale factor (SF) of three, implying that the decimal point is 3-places to left *i.e.*

$$\text{value (X)} = r\text{ value (X)} \times 10^{-SF}$$

SF will always be 3, regardless of the *r*-value of X.

Evaluation :

1. The fixed point numbers have the advantage of being capable of precise storing decimal values, at least those with in a restricted range, wh cannot be done in floating point.
2. The disadvantage of fixed point numbers are that range of values restricted because no exponents are allowed, and their representation memory is wasteful.

(*b*) Floating point real numbers

Specification : A floating point number is based on the idea of scient notation, in which we represent both the mantissa (fractional part) and the expon of a number. The notation 3.2843E-4 is commonly used in printouts to repres 3.2843×10^{-4}.

The main attribute of a floating point number if its type as real in FORTR and Float in C. As with type integer, the values form an ordered sequence from so hardware determined minimum negative value to a maximum value, but the val are not distributed evenly across this range. The operations provided for float point numbers are same as that for integer with following restrictions :

- Boolean operations are sometimes restricted.
- Due to round off issues, equality between two real numbers is ra achieved.

Implementation : In order to use the built-in floating point commands hardware, however, they are generally stored in binary, with some bits for exponent and some for the fraction, as shown in figure 2.11.

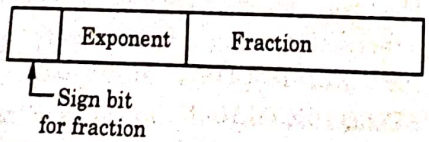


Fig. 2.11. Floating Point Representation.

However, most newer machines use the **IEEE standard 754 format.** Language implementers use whatever representation is supported by the hardware. Most languages include two floating point types, often called **float** and **double.** The float type is the standard size, usually being stored in 4-bytes (32-bits) of memory. The double type is provided for situations where large fractional parts are needed. Double-precision variables usually occupy twice as much storage as float variables and provide at least twice the number of bits of fraction.

The collection of values that can be represented by a floating point type is defined in terms of precision and range. Precision is the accuracy of the fractional part of a value, measured as the number of bits. Range is a combination of the range of fractions, and more importantly, the range of exponents. Figure 2.12 shows the IEEE Floating point standard 754 format for

(*a*) Single precision representation (32 bit, IEEE 1985)

(*b*) Double precision representation (64 bit, IEEE 1985)

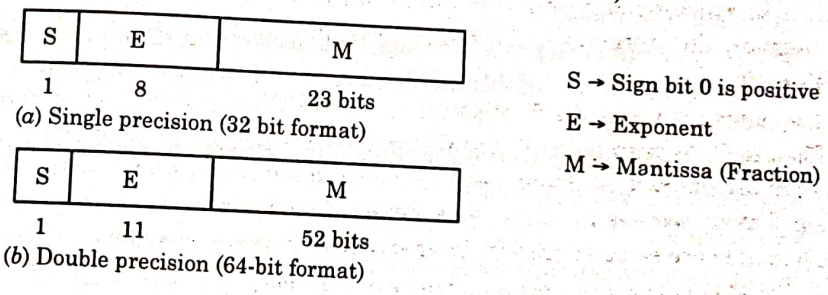


Fig. 2.12. IEEE Floating Point Formats.

In 64-bit format, the exponent is expended to 11 bits giving a range from – 1022 to + 1023, yielding numbers in the range 10^{-308} to 10^{308}. The exponentiation operation is usually software simulated.

3. Subranges :

Specification : Subranges are a special case of basic types because they restrict the range of values of an existing type. For example, the subrange 0..99, restricts attention to the integer values 0 through 99. The values of the underlying type must be integers or they must be of some enumerated type ; subranges of reals are not permitted.

The syntax of a subrange in Pascal is

<constant 1> .. <constant 2>

where <constant 1> and <constant 2> are of the same type, and <constant 1> is less than or equal to <constant 2>. The values in the subrange denoted by low...high are low, low+1, ..., high, if low and high are integers.

e.g. A : 1 ... 10

cause a subrange to take values 1, 2, ... 10.

A subrange type allows the same set of operations to be used as for the ordinary integer type ; thus a subrange may be termed a subtype of the base type integer.

Implementation : Subrange types have two important effects on implementations :

1. **Smaller Storage requirements :** A subrange use a fewer bits than a general integer value *e.g.* the subrange 1 ... 10 requires only 4-bit storage for its representation instead of full integer assuming 16 or 32-bit representation. But the arithmetic operations are software simulated requiring the smallest number of bits for which the hardware implements arithmetic operations.
2. **Better type checking :** A subrange type declaration allows more precise type checking *e.g.* if variable **day** is : day : 1 ... 7 then the assignment

   day : = 0

   is illegal and can be detected at compile time. However, in some cases run time checking is also needed.

Evaluation :

1. Subrange types enhance readability
2. Reliability is increased with subrange types.

4. **Complex Numbers :** A complex number consists of a pair of numbers representing the numbers real and imaginary parts. *e.g.*

   5 + 3i

Here 5 is the real part and 3 is the imaginary part. Each complex data object is represented as a block of two storage location containing a pair of real values. Operations an complex numbers may be software simulated, since they are unlikely to be hardware implemented.

5. **Rational Numbers :** A rational number is the quotient of two integers. The main reason for including rational number data type is to avoid the problem of round off and transaction encountered in floating and fixed-point representation of reals. As a result, it is desirable to represent rational as pairs of integers of unbounded length. Such long integers are often represented using a linked representation.

2.9 ENUMERATIONS

Specification : An enumeration is an ordered list of distinct values. It may be defined as a finite sequence of names written between parentheses (or braces). For example in C

enum day {Mon, Tue, Wed, Thu, Fri, Sat, Sun} ;

makes **day** an enumeration with seven elements.

In Pascal the above C definition represents the enumeration in a separate **"type definition"** and give it a **"type name"** that can then be used to specify the type of several variables as

```
type day = (Mon, Tue, Wed, Thu, Fri, sat, Sun) ;
var
    weekday : day ;
```

Here, the type definition introduces the type name **day**, which may be used wherever a primitive type name such as **integer** might be used.

The values are called enumeration **literals**, shown here as identifiers. They cannot also be used for variable names. In many languages, the Boolean type is essentially a predefined enumeration type : boolean = (false, true) ;

The basic operations on enumeration types are the relational operations (= , <, >, etc), assignment and successor and predecessor.

Implementation : Each value in the enumeration sequence is represented at run time by one of the integers 0, 1, 2, ... The set of values involved is generally small and the values are never negative. The bits enough for representing the range of values are required. For example the example given above have 7 possible values therefore only 3-bits are required to represent these 7 possible values in memory.

The operations on enumeration types are generally implemented using hardware provided operations. For example, relational operations such as =, < and > may be implemented using the corresponding hardware primitives that compare integers.

Evaluation :

1. Enumeration types provide advantages in both readability and reliability.
2. Named values are easily recognized, whereas coded values are not.
3. The **enum** types of C and C++ are not included in Java.

2.10 BOOLEANS

Specification : Boolean types are perhaps the simplest of all types. Their range of values has only two elements, one for true and one for false. In Pascal and Ada, the Boolean data type is considered simply a language – defined enumeration :

```
type Boolean = (false, true) ;
```

which both defines the names true and false for the values of the type and defines the ordering false < true.

The most common operations on Boolean types include assignment as well as :

and : Boolean × Boolean → Boolean

or : Boolean × Boolean → Boolean

not : Boolean → Boolean

Other operations include equivalence, exclusive, implication, nand, nor and short-circuit evaluation for the boolean operators.

Implementation : A Boolean value could be represented by a single bit, but because a single bit of memory is difficult to access efficiently on many machines they are often stored in the smallest efficiently addressable cell of memory, typically a byte. There are two options.

(a) A particular bit is used for the value 0 = false, 1 = true and the rest of the byte or word is ignored.

(b) A zero value in the entire storage unit represents false, and any other non zero value represents true.

Evaluation :

1. Boolean types are often used to represent switches or flags in programs.
2. The use of Boolean types is more readable.
3. C does not have an explicit Boolean data type, integer is used.

True is any non zero value, and false is 0.

2.11 CHARACTERS

Specification : Most data are input and output in character form. Character data are stored in computers as numeric codings. A character data type provides data objects that have a single character as their value. ASCII (American Standard Code for Information Interchange) is the most common and is often supported by hardware. For 7-bit ASCII, the codes from 0 to 127 represents 128 different characters which includes printable characters (alphanumeric characters) as well as

a number of control characters, useful for printer and screen control. 8-bit codes provide extended character sets in the range 128 to 255. The Java language supports a 16-bit code called Unicode in order to support more non-English characters. The numeric ordering of the characters in the character set is called the **collating sequence** for the character set. The collating sequence is important because it determines the alphabetical ordering given to character strings by the relational operations.

The operations on character data include only the relational operations, assignment and sometimes operations to test whether a character value is one of the special classes "letter", "digit" or "special character".

Implementation : For character values, the hardware or operating system character codes are used. The reason for the choice is simple.

(a) If the hardware storage representations are used, then the basic operations on data of that type may be implemented using the hardware provided operations.

(b) If the hardware storage representations are not used, then the operation must be software simulated, and the same operations will execute much less efficiently.

Ordinarily a language implementation uses the same storage representation for characters. Since characters arrive from the Input-Output system in the hardware-supported representation, the language implementation may have to provide appropriate conversions to the alternative character – set representation or provide special implementation of the relational operations that take account of differences in the collating sequence.

KEY POINTS TO REMEMBER

- A **data object** represents a place where data values may be stored and later retrieved.
- A data object can be **elementary** or **structured**.
- The data object that exist during the program execution can be system defined or programmer defined.
- A **name** is a string of characters used to identify some entity (or data object) in a program.

- A **variable** is defined as a data object that is defined and named by a programmer explicitly in a program.
- A **constant** is defined as a data object that is bound to a value permanently during its lifetime.
- A **constant** in a program can be either literal constant or programmer defined (manifest constant).
- A data type is defined as the class of data objects together with a set of operations for creating and manipulating them.
- The data types that a programming language provides can be broadly classified into **primitive** and **non-primitive** data types.
- The basic elements of a specification of a data type are **attributes**, **values** and **operations**.
- The basic elements of an implementation of a data type are storage representation and implementation of operations.
- A data object is said to be **elementary data object** if the value it contains can be manipulated as a single unit.
- The **signature** of an operation specifies the number, order and data type of the arguments in the domain of an operation as well as the order and data type of the resulting range. The signatures of an operation are specified using the notation :

$$op\_nme : arg_1\ type \times arg_2\ type \times \cdots\cdots \times arg_n\ type \rightarrow result\_type$$

- The **operations** defined for data objects of a given type may be implemented :
  - — Directly as a hardware operation
  - — As a function subprogram or procedure
  - — As an inline code sequence
- *A* ***declaration*** in a program refers to a statement that provides the information about the name and type of data objects to the programming language translator.
- *A* ***declaration*** can be either **explicit** or **implicit**.
- *The* ***type checking*** is the activity of ensuring that the operands of an operator are of compatible types. A compatible type is one that is either legal for the operator or is allowed under language rules to be implicitly converted by compiler-generated code to a legal type. This automatic conversion is called a **coercion.**

- The type checking may be done at compile time (called static type checking) or at run time (called dynamic type checking).
- A type system is **strong** if it accepts only safe expressions.

EXERICISE

1. What do you mean by a data object ? Differentiate between data object, variables and constants.
2. Define the term data type. Distinguish between primitive and non-primitive data types.
3. Explain the specification and implementation of elementary data types.
4. What do you mean by a declaration ? What is the purpose of a declaration?
5. Define the term type checking and type conversion. Also explain static and dynamic type checking.
6. Explain the concept of numeric data types in detail.
7. Write short notes on :
   (*a*) Assignment and initialization
   (*b*) Implicit and explicit type conversion
   (*c*) Variable and constant
   (*d*) Data object and data type

CHAPTER

3 SYNTAX AND SEMANTICS

3.1. INTRODUCTION

A language is a set of valid sentences. The main purpose of a language is to express meaning by means of sound (or gesture). Similarly, programming languages implementors must be able to determine how the expressions, statements and program unit of a language are formed, and also their intended effect when executed. The validity of the sentences comprising a programming language can be broken down into two things namely syntax and semantics. These two terms are discussed separately.

3.1.1. SYNTAX

The term **syntax** refers to grammatical structure i.e. syntax of a programming language is the form of its expressions, statements and program units. The primary purpose of the syntax is to provide a notation for communication between the programmer and the programming language processor. There are some important points regarding syntax :

1. The syntax of a programming language is the arrangement of words and elements in a sentence to show their relationship i.e. it describes the sequence of symbols that make up valid programs. e.g. in C language a = b + c represents a valid sequence of symbols whereas ab +− does not represent a valid sequence of symbols for a C program.
2. The syntax of a programming language is the set of rules which determines if the sentence is well formed or not. It also tells how to form expressions, statements and program units that look right e.g. the syntax of C **if** statement is

   if (< expr >) < statement >
3. The syntax of a programming language is defined as a set of rules which specify the composition of program from letters, digits and other special characters. Using the syntax rules, we can tell whether, a sentence

legal or not e.g. keywords, identifiers, numbers, operators are the words of the C language. The C syntax tell us how to combine such words to construct the well formed statements and program e.g. the syntax of a sample C program is illustrated in program shown in **fig. 3.1**

```
# include < stdio.h >
void  main ( )
{
        prinff (" Hello world!/n" );
        }
```

Fig. 3.1. A Sample C program Syntax.

3.1.2. SEMANTICS

The term semantics refer to the meaning of the vocabulary symbols arranged with that stricture. In a well-designed programming language, semantics should follow directing from syntax; i.e. the form of a statement should strongly suggest what the statement is meant to accomplish. Therefore, semantics of a programming language describe the relationship between the syntax and the model of the computation.

The semantics is concerned with the interpretation or understanding of programs and how to predict the outcome of program execution. Semantics can be thought of as a function which maps syntactical constructs to the computational model i.e.

Semantics : Syntax → Computational model

This approach is called **syntax-directed semantics**. Some of examples of programming language semantics are

1. Consider the syntax of a C **if** statement

   if (< expr >) < statement >

The semantics of this statement form is that if the current value of the expression is true, the embedded statement is selected for execution.

2. Consider the following two declarations :

   var a : **array** [1..10] **of integer** ;

   int a [10] ;

Both of these declarations refers to different syntax in Pascal and C respectively but same semantics. The semantics of both of these declaration is to declare an array of ten integer elements and the elements of the array may be referenced by index.

Note : If two sentences are syntactically valid then it does not imply that they are semantically valid also e.g. the grammatical sentence "Cow flow supremely" is grammatically ok (subject verb adverb) in English, but makes no sense.

3.2. THE GENERAL PROBLEM OF DESCRIBING SYNTAX

The syntax of a programming language describes the structure of programs without any consideration of their meaning. The syntax of all programming languages (like natural languages e.g. English) is defined by two set of rules :

1. Lexical Rules
2. Syntactic Rules

These rules are explained in brief as :

1. **Lexical Rules :** The lexical rules include the following terminology.
   - *(i)* **Input Alphabet (Σ) :** It is defined as a set of all input symbols used in making valid sentences (or words) of a language e.g. in case of English language the Σ = {a,b,.....,z}. The alphabet for machine language is Σ = {0,1}.
   - *(ii)* **Alphabet* (Σ^*) :** It is defined as the set of all possible strings or words that can be derived from a given alphabet e.g. if
     Σ = {0,1}
     ⇒ Σ^* = {∧,0,1,00,01,10,11,.....}
   - *(iii)* **Language (L) :** It is defined as a set of strings or words from Σ^* satisfying a given criteria e.g. If Σ = {0,1} and L = {$0^n 1^n$ / n≥0} then L = {∧, 01,0011,.....}
     Clearly $L \subseteq \Sigma^*$

The above discussion implies that the lowest level of description of rogramming language syntax include its identifiers. literals, operators, keywords nd other special characters. These units are known as **Lexemes.** (input symbols om alphabet). The description of lexemes is given by lexical specification.

A **token** of a language is a category of its lexemes. For example, an identifier is oken that can have lexemes such as position, sum, area etc. In general, a lexeme n instance of a token. e.g. consider the following C statement:

```
total = 2 * sum + 5 ;
```


The various Lexemes and tokens ax

Lexemes	Tokens
total	identifier
=	Rel_operator
2	literal
*	mult_operator
Sum	identifier
+	add–operator
5	literal
;	Semicolon (Punctuation symbol)

The Lexical rules are implemented using language Recognizers.

2. **Syntactic rules :** The syntactic rules describe how the lexical units (tokens) are grouped into meaningful strictures. e.g. consider the following expression

a+/b

The lexical rules transform it into

id* + / *id where ***id*** represents identifier.

The syntactic rules reflect that it is incorrect syntactically because the presence of two consecutive binary operators is a meaningless construct as each binary operator requires two operands which are not present here. The syntactic rules also impose a tree like hierarchical structure on the meaningful syntactically valid programming language structures. The syntactic rules are implemented using language generators.

3.2.1 LANGUAGE RECOGNIZERS AND LANGUAGE GENERATORS

The programming language can be formally defined in two distinct ways :

(*a*) Language Recognition and

(*b*) Language Generation.

(*a*) Language Recognition : Consider a Language L defined over an alphabet (Σ). The language recognition method constructs a mechanism R, called a **recognition device** (generally a **finite automata**). The recognition device R is capable of answering whether a given input string was or was not accepted by R. If all the strings comprising a language L are accepted by the recognition device R then

the recognition device R is said to recognize a language. For example, consider a Language L defined over alphabet $\Sigma = \{a,b\}$ as

$L = \{x \mid x \in \{a,b\}^* \text{ and ends with abb}\}$.

The Language Recognition device R is

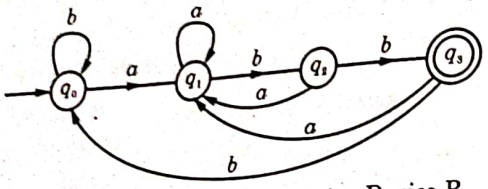


Fig. 3.2. A Language Recognition Device R.

The language Recognition device shown in fig. 3.2 recognize a Language consisting of all the strings ending with abb. Such Language Recognition devices are called **finite automata.**

(b) Language generation : A **language generator** is a device that is used to generate the strings or words of a given language. A Language generation method constructs a language generator G and is capable of generating a string of language by a process called **derivation**. For example consider a Language L over alphabet $\Sigma = \{a, b\}$

$L = \{a^n b^n \mid n \geq 0\}$

$w = a\ abb$

$G : \{S \rightarrow aSb / \wedge\}$

Derivation :

$S \Rightarrow asb$

$\Rightarrow aasbb$

$\Rightarrow aabb$

i.e. $\boxed{S \overset{*}{\Rightarrow} aabb}$. It means we have derived the string aabb starting from start symbol S using the rules of Language Generator G in finite number of steps.

3.3 FORMAL METHODS OF DESCRIBING SYNTAX

This section deals with various formal methods of describing programming nguage syntax. The formal definition of the syntax of a programming language is ually called a **grammar**, as an analogy with the common terminology for natural

languages. A grammar consists of a set of rules (called productions) that specify the sequence of characters (or lexical items) that form allowable programs in the language being defined. The various formal methods for describing syntax are :

1. BNF grammars.
2. Context free grammars
3. Extended BNF
4. Syntax Graphs
5. Finite state automata
6. Regular Grammars and regular expressions
7. Push Down Automata.

The methods are discussed in detail.

1. BNF Grammar : The Backus–Naur Form(BNF) was developed in the late 1950s by the linguists **John Backus** and **Peter–Naur** for the syntactic description of ALGOL. BNF is a very natural notation for describing syntax. A BNF grammar G is defined as

$G = (V, T, S, P)$

Where V = Set of abstraction in a BNF description or grammar. The abstractions are often called variables, non-terminals or syntactic categories

T = Set of terminal symbols and generally includes some mixture of tokens and lexemes

S = A special non terminal (abstraction) called the start symbol.

P = A set of productions or rules.

The above parameters are illustrated in more detail as :

1. Abstractions : BNF uses abstractions for syntactic structures. The non terminals or variables are enclosed between < and > . The abstractions are actually the things having definitions. The examples of valid abstractions are < assign >, < program > etc.

2. Productions or rules : The productions or rules in BNF have the general form as $\boxed{A ::= \alpha}$ or $\boxed{A \rightarrow \alpha}$. The symbols :: = and → are used interchangeably.

The symbol on the left side of :: = or →, called the left hand side (LHS), is the abstraction being defined. The text to the right of the :: = or → is the definition of the L.H.S. It is called the right hand side (RHS) and consist of mixture of tokens, lexeme and references to other abstractors. e.g. consider a production

< assign > → < id > : = < expr >

This rule specifies that the abstraction < assign > is defined as an instant of the abstraction < id > follower by lexeme : = , followed by an instance of the abstraction < expr > e.g. one valid sentence whose syntactic structure is described by the rule is

$id_1 := id_2 + id_3$

Note : 1. BNF is **meta-language** i.e. a language used to describe another language.

2. BNF is simple and sufficiently powerful to describe the great majority of the syntax of programming languages.

3. The BNF allows the implementation of recursive rules i.e. a rule may itself be used in that rule in order to specify repetition. e.g. the recursive rule

< unsigned_integer > :: = < digit > / < unsigned_integer > < digit >

defines an unsigned integer as a sequence of < digit >. The first alternative < digit > defines an unsigned integer as a single digit , while the second alternative adds a second digit to this initial digit, a digit on to the second and so forth.

3.3.1. DERIVATIONS AND PARSE TREES

Beginning with the start symbol of the BNF grammar, the process of generating the sentences of a language through a sequence of applications of the rules is called **derivation process** or **parsing.** Each successive string in the sequence is derived from the previous string by replacing one of the non terminals with one of that non terminal's definition. Each string in the derivation is called a **sentential form.** The process of derivation is as shown below

$$S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \ldots\ldots\ldots\ldots \Rightarrow w_n \Rightarrow w$$

The above sequence represents the derivation of the sentence w staring from S. Here w_i (for i = 1 to n) are called **sentential forms.** The symbol ⇒ mean **"derives"** These are basically two ways by which a derivation can take place

1. **Left Most Derivation (LMD) :** A derivation sequence in which at each step the leftmost non terminal is replaced always is called a leftmost derivation (LMD).
2. **Right Most Derivation (RMD) :** A derivation sequence in which at each step the rightmost nonterminal is replaced always is called a right most derivation (RMD).

The derivation sequence generated using either LMD or LMD can be represented graphically in the from of a hierarchical structure called a

parse tree or derivation tree. A derivation tree for a BNF grammar G has the following properties :

(*i*) The root is labeled with the start symbol S.

(*ii*) Every interior node is labeled with a nonterminal symbol from V.

(*iii*) If a node has a label A in V and its children are labeled (from left to right) $X_1, X_2, \ldots\ldots X_n$, then P must contain a production of the form

$$A \rightarrow X_1, X_2, \ldots\ldots X_n$$

(*iv*) Every leaf is labeled with a terminal symbol from T.

Note : Every subtree of a parse tree describes one instance of an abstraction in the statement.

Example 3.1. Write a BNF grammar for describing Lists.

Solution. Variable length lists are often written in mathematics using an ellipse (.....) ; e.g. 1,2,..... But BNF does not include an ellipse. Therefore, the recursive rules are used to describe the variable length lists using BNF grammar. The following recursive rules are used

< id_list > → id

| id, < id_list >

This means that <id_list > (identifier list) is defined either as a single identifier or an identifier followed by another instance of the abstraction <id_list>

Example 3.2. Describe in English, the language defined by the following grammar

< S> → <A > <B> <C>

<A> → *a*<A> | *a*

<B> → *b* <B> | *b*

<C> → *c* <C> | *c*

Solution. The above grammar defines a language that consists of all the sentences in which an instance of <A>, followed by an instance of <B>, followed by an instance of <C>. In turn <A>, is defined either as an **a** or **a** followed by an instance of <A>. Similarly <B>, is defined as either as a **b** or **b** followed by an instance of <B>. <C> is defined either as a **c** or a **c** followed by an instance of <C>. This implies that above grammar defines he language over an Σ = { *a,b,c,* } which consists of all the strings with some finite number of a's, followed by same finite number of b's and some finite number of c's.

Example 3.3. Consider the grammar of example 3.2. Show a parse tree and a leftmost derivation for the string

aaabbbccc

Solution : Leftmost derivation,

Parse Tree

<S> ⇒ <A> <B> <C>
⇒ a <A> <B> <C>
⇒ aa<A> <B> <C>
⇒ aaa<B> <C>
⇒ aaab <B> <C>
⇒ aaabb <B> <C>
⇒ aaabbb <C>
⇒ aaabbbc <C>
⇒ aaabbbcc <C>
⇒ aaaabbbccc

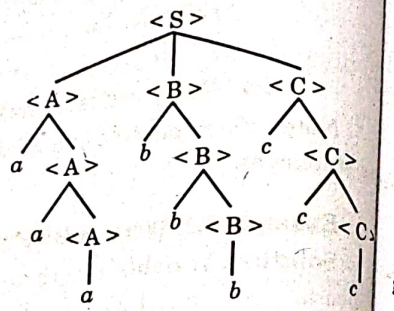


Fig. 3.3 A Passe Tree

Example 3.4.consider the following grammar

<S> → <A> a <B> b
<A> → <A> b | b
<B> → a <B> | a

which of the following sentences are in the language generated by this grammar ?

(a) baab (b) bbbab (c) bbaaaaa (d) bbaab.

Solution :The general pattern of the strings being derived from this grammar includes b.....b a a .....a b i.e. all the strings must start with b and containing some finite number of b's (at least one) followed by an a, followed by some finite number of a's (at least one), followed by b.

(a) baab : Clearly this string matches with given pattern and hence generated as

<S> ⇒ <A> a <B> b
⇒ ba <B> b
⇒ b a a b

(b) bbbab : This string is not generated as it contains only one a but minimum 2 a's are required to match with given pattern.

(c) bbaaaaa : This string is not generated as it does not end with b

(d) bbaab : Clearly, this string matches with given pattern and hence generated as

<S> ⇒ <A> a <B> b
⇒ <A> ba <B> b
⇒ b a a <B> b
⇒ bb aa b.

Example 3.5 Consider the following grammar.

<S> → *a* <S> *c* <B> | <A> | *b*
<A> → *c* <A> | c
<B> → *d* | <A>.

which of the following sentences are in the language generated by this grammar ?

(a) abcd (b) acccbd
(c) acccbcc (d) acd
(e) accc.

Solution : (*a*)The sentence is generated as

S ⇒ *a* <S> *c*<B>
⇒ abc <B>
⇒ abcd.

(*b*) The sentence accbd is not generated .

(*c*) The sentence acccbcc is not generated.

(*d*) The sentence acd is not generated.

(*e*) The sentence is generated as.

S ⇒ *a* <S> *c*<B>
⇒ *a*<A> *c* <B>
⇒ *acc* <B>
⇒ *acc* <A>
⇒ *accc.*

Example 3.6. Write a grammar for assignment where right hand side consist of arithmetic expressions with multiplication and addition operators and parentheses. The identifiers for use are A, B and C respective. Also shown a parse tree for A := B× (A + c).

Solution : An assignment statement is defined as a statement in which the value of an expression is assigned to an identifier. Therefore, the BNF grammar consist of

<assign> → <id> : = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>.

The LMD for A = B* (A+C) is given as.

<assign> ⇒ <id> : = <expr>
⇒ A : <expr>
⇒ A : = <id> * <expr>
⇒ A : = B * <expr>
⇒ A : = B * (<expr>)
⇒ A : = B * (<id> +<expr>)
⇒ A : = B * (A + <expr>)
⇒ A : = B *(A + <id>)
⇒ A : = B * (A + C)

The corresponding parse bee is as shown in fig 3.4

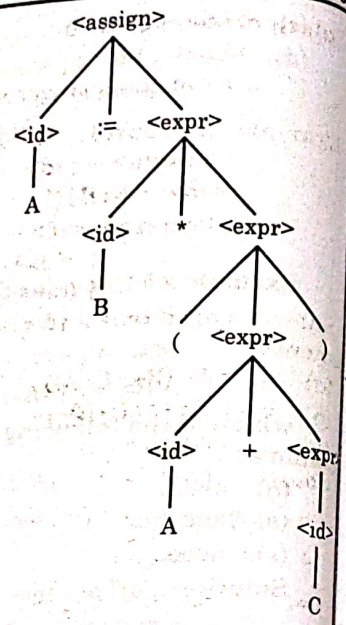


Fig. 3.4. A Parse Tree

Example 3.7. Describe, in English, the language defined by th following grammar.

<program> → **begin** <stmt_list> **end**
<stmt–lest> → <stmt>
| <stmt>; <stmt_list>
<stmt> → <var> : = <expression>
<var> → A | B | C
<expression> → <var> + <var>
| <var> – <var>
| <var>.

Solution : The language defined by above grammar has only one statement form : assignment. A program consists of the special word **begin,** followed by a list of statements separated by semicolons, followed by the special word **end.** An expression is either a single variable, or two variables separates by either a + or - operator. The only variable names in this language are A, B and C respectively.

Example 3.8. Write a grammar for the language consisting of strings that have n copies of the letter a followed by the same number copies of the letter b, where n>0. For example, the string ab, aaaabbbb, and aaaaaaaabbbbbbb are in the language but a, abb, ba and aaabb are not.

Solution. <S> → a <S> b
| ab

The above grammar generates the required strings.

Example 3.9. Consider the grammar for simple assignment statement
<assign> → <id> : = <expr>
<id> → A | B | C
<exper> → <id> + <expr>
| <id>* <expr>
| (<expr>) | <id>

Show a parse tree and a leftmost derivation for each of the following atements

(a) A : = A* (B+ (C*A))
(b) B : = C* (A*C+B)
(c) A : = A* (B+(C))

Solution (a) A : = A* (B+(C*A))

<assign> → <id> : = <expr>
⇒ A : = <expr>
⇒ A : = <id> * <expr>
⇒ A : = A* <expr>
⇒ A : =A* (<expr>)
⇒ A : =A* (<id>+ <expr>)
⇒ A : =A* (B + expr>)
⇒ A : =A*(B + (<expr>))
⇒ A : =A* (B+ (<id>*<expr>))
⇒ A := A*(B+(C*<expr>))
⇒ A := A*(B+(C* <id>))
⇒ A := A*(B+(C* A))

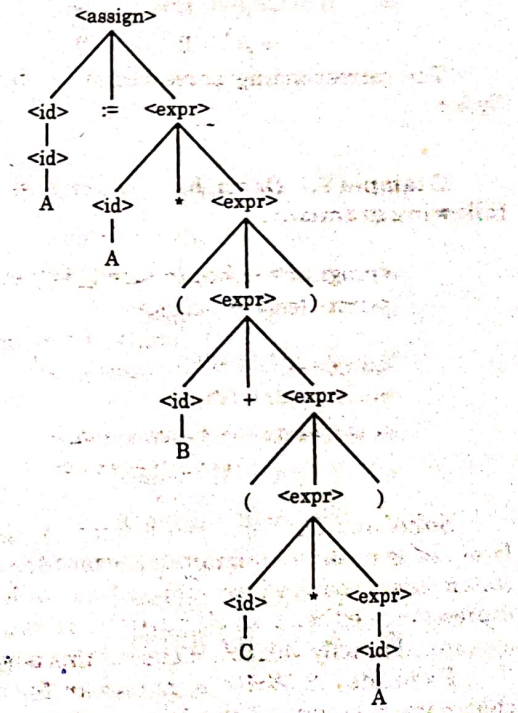


Fig. 3.5. A Parse Tree for A := A * (B + (C * A))

(b) B : = C * (A * (C+ B)

<assign> → <id> : <expr>

⇒ B: = <expr>

⇒ B : <id> * = <expr>

⇒ B := C* <expr>

⇒ B := C* (<expr>)

⇒ B := C* (<id>*expr>)

⇒ B := C*(A*expr>))

⇒ B : =C* (A*(<id>+ <expr>))

⇒ B : C*(A* C +<expr>))

⇒ B := C*(A*C+ <id>)

⇒ B := C*(A*C+ B)

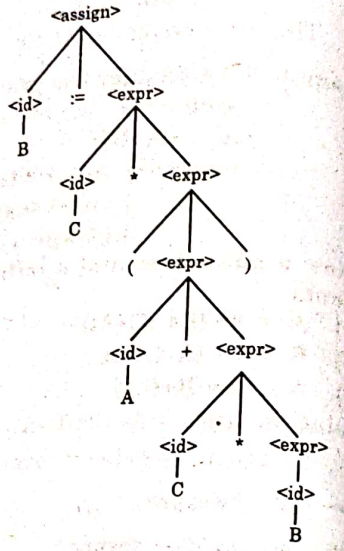


Fig. 3.6. A Parse Tree for B := C * (A * (C + B))

(c) A : = A* (B+(C))

assign> → <id> : = <expr>

⇒ A : = <expr>

⇒ A : <id> * <expr>

⇒ A :=A* <expr>

⇒ A : =A* (<expr>)

⇒ A : =A* (<id>+ <expr>)

⇒ A : =A* (<B + expr>)

⇒ A : =A*(B + (expr>))

⇒ A : =A* (B+ (<id>*<expr>))

⇒ A : =A* (B+ (<id>))

⇒ A : =A* (B+ (C))

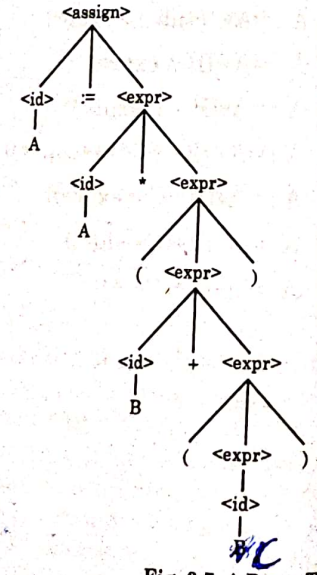


Fig. 3.7. A Parse Tree for A := A * (B + (C))

Example 3.9. Consider the following grammar.

<assign> → <id> : = <expr>

<id> → A | B | C

<exper> → <id> + <term>

| <term>

<term> → <term> * <factor>

| <term>

<factor > → <term>

| <id>.

Show a parse tree and a leftmost derivation for each of the following statements

(a) A : = (A+ B)*C

(b) A: = B+ C+ A

(c) A : = A*(B+ C)

(d) A : = B* (C* (A+ B))

Solution (a) A : = (A +B) *C

<assign> → <id> : = <expr>

⇒ A : = <expr>

⇒ A : <term>

⇒ A := <term>*<factor>

⇒ A : = <factor>*<factor>

⇒ A : = (<expr>)*<factor>

⇒ A : = (<expr> +<term>)*<factor>

⇒ A : = (<term> +<term>)*<factor>

⇒ A : = (<factor> +<term>)*<factor>

⇒ A := (<id> +<term>)*<factor>

⇒ A := (A+<term>)*<factor>

⇒ A := (A+<factor>)*<factor>

⇒ A := (A+<id>)*<factor>

⇒ A := (A+B)*<factor>

⇒ A := (A+B)*<id>

⇒ A := (A+B)*C

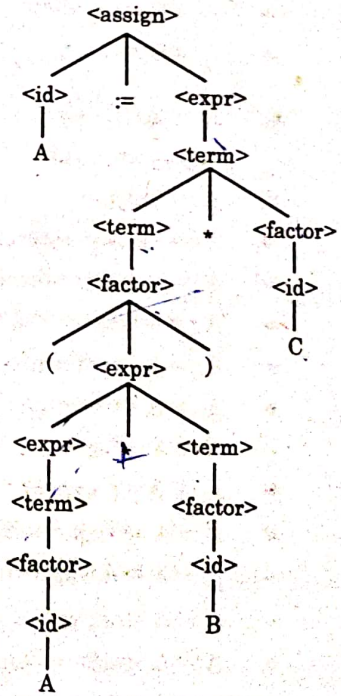


Fig. 3.8. A parse tree for A : = (A + B) * C

(b) A : = B+ C + A

<assign> → <id> : = <expr>

⇒ A : = <expr>

⇒ A : = <expr> +<term>

⇒ A : = <expr> +<term>+ <term>

⇒ A : = <term> +<term>+ <term>

⇒ A : = <factor> +<term>+ <term>

⇒ A : = <id> +<term>+ <term>

⇒ A := B +<term>+ <term>

⇒ A := B +<factor>+ <term>

⇒ A := B +<id>+ <term>

⇒ A := B + C + <term>

⇒ A := B + C + <factor>

⇒ A := B + C + <id>

⇒ A := B + C + A

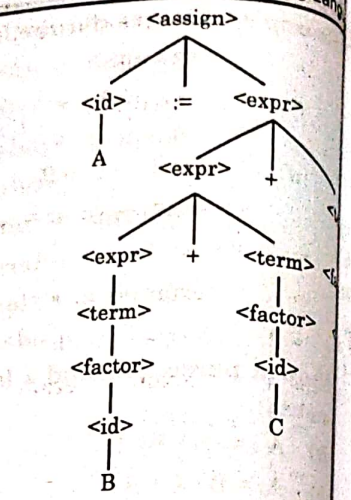


Fig. 3.9. A parse tree for A : = B + C + A

(c) A : = A*(B+ C)

<assign> → <id> : = <expr>

⇒ A : = <expr>

⇒ A : = <expr> *<term>

⇒ A := <term>* <term>

⇒ A : = <factor>* <term>

⇒ A : = <id>*<term>

⇒ A : = A *<term>

⇒ A : =A *<factor>

⇒ A : = A * (<expr>)

⇒ A := A *(<expr>+ <term>)

⇒ A := A *(<term> + <term>)

⇒ A := A *(<factor> + <term>)

⇒ A := A *(<id> + <term>)

⇒ A := A *(B+ <term>)

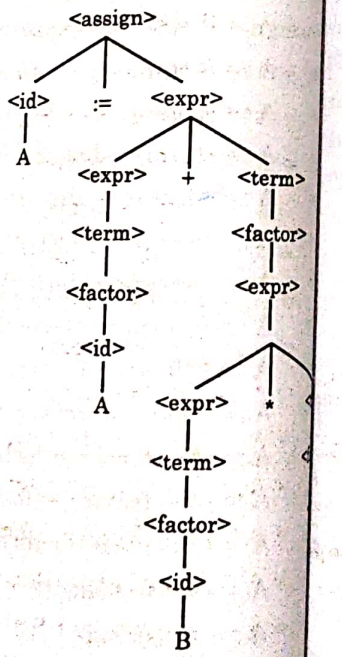


Fig. 3.10. A parse tr
A : A*(B + C)

⇒ A := A *(B+ <factor>)

⇒ A := (B+ <id>)

⇒ A : = A*(B+ C)

(d) A : = B*(C * (A+ B))

<assign> → <id> : = <expr>

⇒ A : = <expr>

⇒ A : = <expr> *<term>

⇒ A := <term>* <term>

⇒ A : = <factor>* <term>

⇒ A : = <id>*<term>

⇒ A : = B *<term>

⇒ A : =B *<factor>

⇒ A : = B * (<expr>)

⇒ A := B *(<expr>* <term>)

⇒ A := B *(<term> * <term>)

⇒ A := B *(<factor> * <term>)

⇒ A := B *(<id> * <term>)

⇒ A := B *(C* <term>)

⇒ A := B *(C* <factor>)

⇒ A : + B * (C*(<expr>)

⇒ A : = B *(C* (<expr> + <term>))

⇒ A : = B *(C* (<term> + <term>))

⇒ A : = B*(C*(<factor> + <term>))

⇒ A : = B * (C(<id> + <term>))

⇒ A : = B * (C* (A + <term>))

⇒ A : = B * (C* (A + <factor>))

⇒ A : = B * (C* (A + <id>))

⇒ A : = B * (C* (A + B))

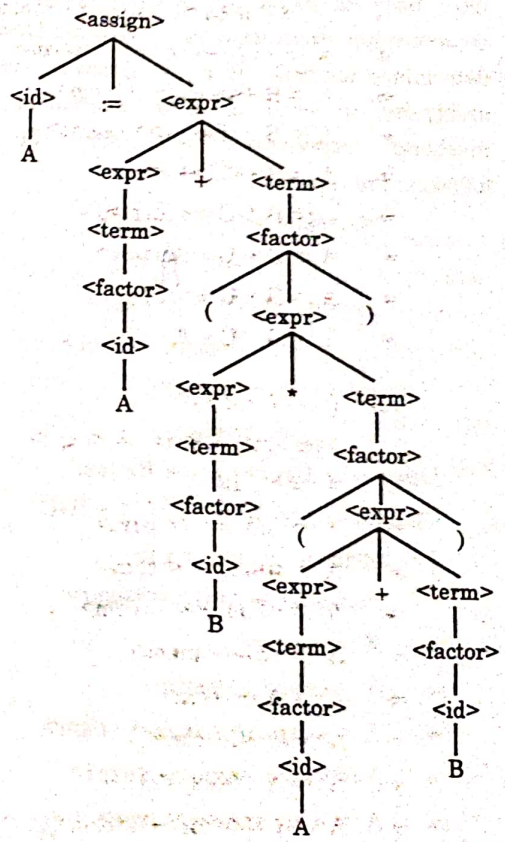


Fig. 3.11. A parse tree for A : = B*(C*(A+B))

3.3.1.1 AMBIGUITY

A BNF grammar is said to be ambiguous if there exits some string w which generates two distinct derivation trees. When there are two or more left most derivations of a string in a given grammar G, it results in two distinct derivation trees for the same sentence, the grammar is said to be ambiguous.

The syntactic ambiguity of language structures is a problem because compilers often base the semantics of those structures on their syntactic from. If a language structure has more then one derivation tree, the meaning of the structure can not be determined uniquely. Here, we illustrate the concept of ambiguity and the solution to ambiguity in case of BNF grammars involving expressions. Consider the following grammar.

<assign> → <id> : = <expr>

<id> → A | B | C

<expr> → <expr> + <expr>

| <expr> * <expr>

| (<expr>)

| <id>.

Operator Precedence Rules:

Consider a string W, A : = A + B*C

The leftmost Derivation is

<assign> → <id> : = <expr>

⇒ A : = <expr>

⇒ A : <expr> * <expr>

⇒ A := <expr>+ <expr>* <expr>

⇒ A := <id>+ <expr>* <expr>

⇒ A := A + <expr>* <expr>

⇒ A := A + <id>* <expr>

⇒ A := A + B* <expr>

⇒ A := A + B* <id>

⇒ A := A + B* C

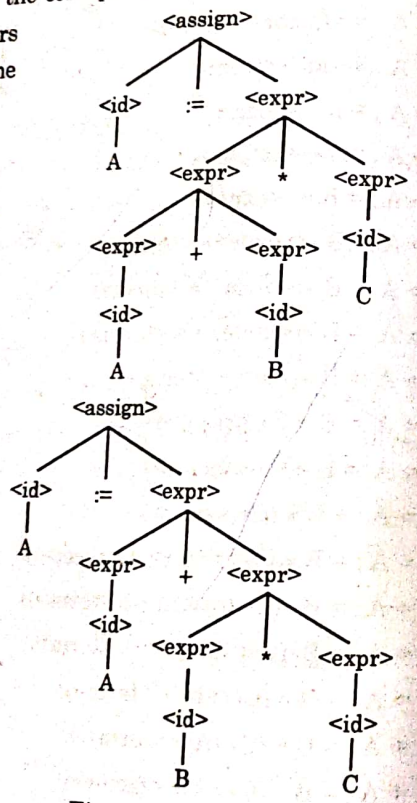


Fig. 3.12. Two distinct parse tree A : = A + B * C

There exist another LMD corresponding to same string

<assign> → <id> : = <expr>

⇒ A : = <expr>

⇒ A : <expr> + <expr>

⇒ A := <id>+ <expr>

⇒ A := A + <expr>

⇒ A := A + <expr>* <expr>

⇒ A := A + <id>* <expr>

⇒ A := A + B* <expr>

⇒ A := A + B* <id>

⇒ A := A + B* C

Fig. 3.12 reflects the fact the fait that there are two distinct parse trees for the string A : = A + B* C. The first tree represents the fact that '+' operator appears at a lower level than '*' operator. The second tree represents the fact that '*' operator appears at a lower level than '+' operator. The general rule is that an operator appearing at a lower level has higher priority than the operator at a higher level. Therefore, the first tree interpretes the meaning as (A +B) *C, while the second tree interprets the meaning as A + (B*C). According to the rules of operator precedence the operator '*' has higher priority than the operator '+'. Therefore the second tree represents the correct precedence of operators and is a valid parse tree.

Associativity of operators : Consider a string A : = A + B + C.

The LMD for the given string consist of

<assign> → <id> : = <expr>

⇒ A : = <expr>

⇒ A : <expr> + <expr>

⇒ A := <id>+ <expr>

⇒ A := A + <expr>

⇒ A := A + <expr>+<expr>

⇒ A := A + <id>+ <expr>

⇒ A := A + B+<expr>

⇒ A := A + B+ <id>

⇒ A := A + B+ C

There exists another LMD corresponding to the same string.

<assign> → <id> : = <expr>
⇒ A : = <expr>
⇒ A : <expr> + <expr>
⇒ A := <expr>+ <expr>+ <expr>
⇒ A := <id>+ <expr>+ <expr>
⇒ A := A + <expr>+<expr>
⇒ A := A + <id>+ <expr>
⇒ A := A + B+<expr>
⇒ A := A + B+ <id>
⇒ A := A + B+ C

Here both the operators appearing in the string are '+' operators. Therefore, the rule of operator precedence does not apply here. When all the operators appearing in a string belongs to the same precedence level then the rules of associativity of operators comes into existence. The operator. +,−, and / are left associative i.e. evaluator starts from left to right. The exponential operator is right associative i.e. evaluation starts from right to left therefore, according to the rules of operator associativity for a given string, the leftmost '+' has higher pointy then right '+'. The left '+' appears at a lower level in second parse tree reflecting its higher priority then other '+' operator. Therefore the second tree represent the correct associativity of operators and is a valid parse tree.

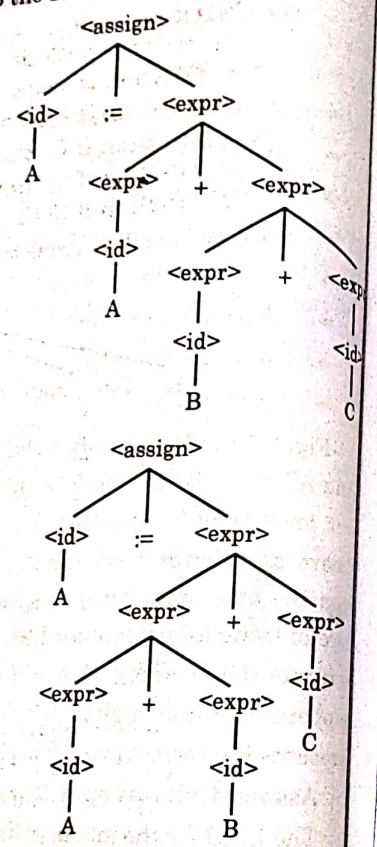


Fig. 3.13. Two distinct parse tree the string A : = A + B + C

Example 3.10 :Explain the concept of ambiguity for if_then_else grammar.

Solution: The BNF grammar rules for if_ then_ else statement is given as

<if_ stmt> → **if** <logic expr> **then** <stmt>
| **if** <logic_ expr> **then** <stmt> **else** <stmt>

The grammar written above is ambiguous. To check the ambiguity consider the following string.

if C_1 then if C_2 then S_1 else S_2

The LMD is given as

<if_ stmt> → **if** <logic _ expr> **then** <stmt> **else** <stmt>
⇒ **if C_1 then if** <logic _ expr> **then** <stmt> **else** <stmt>
⇒ **if C_1 then if** <Stmt> **else** <stmt>
⇒ **if C_1 then if C_2 then** <stmt> **else** <stmt>
⇒ **if C_1 then if C_2 then S_1 else** <stmt>
⇒ **if C_1 then if C_2 then S_1 else S_2**

The corresponding parse tree id given as in figure 3.14

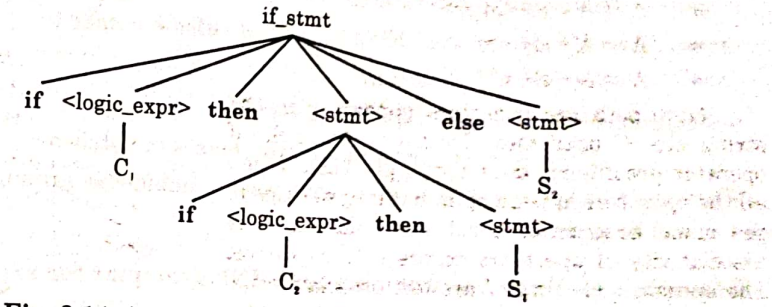


Fig. 3.14. A parse tree for "if C_1 then if C_2 then S_1 else S_2".

There exists another LMD for the same string which is given as

<if_ stmt> → if <logic _ expr> then <stmt>
⇒ if C_1 then <stmt>
⇒ if C_1 then if <logic _ expr> <stmt> else <stmt>
⇒ if C_1 then if C_2 then<stmt> else <stmt>
⇒ if C_1 then if C_2 then S_1 else <stmt>
⇒ if C_1 then if C_2 then S_1 else S_2

The corresponding parse tree is shown in gig. 3.15.

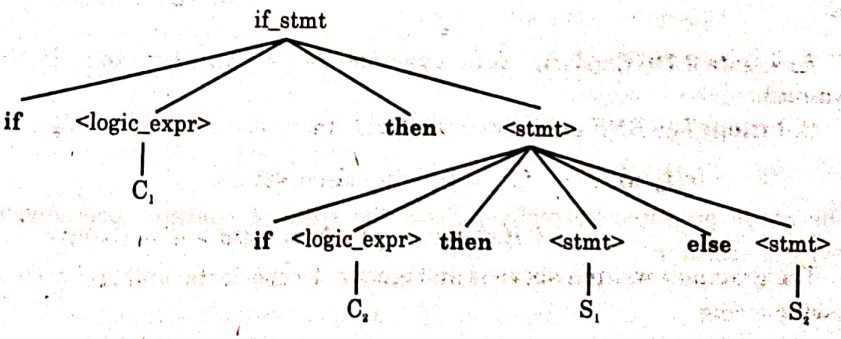


Fig. 3.15. Another parse tree for "if C_1 then if C_2 then S_1 else S_2".

As there exists two distinct parse trees for the same string. Therefore the grammar is ambiguous.

We will now write an unambiguous grammar that describes this **if** statement. The rule for **if** statement is that an 'else' clause should match with nearest unmatched 'then'. Therefore, between a **then** and its matching **else**, there cannot be an **if** statement without an **else**. The 'else' is matched with nearest 'then' in the parse tree of figure 3.15 representing the fact that it is the correct parse tree. Based on this idea the unambiguous grammar for if then else is

<stmt> → <matched> | <unmatched>

<matched> → **if** <logic _ expr> **then** <matched> **else** <matched>

| any non– if statement.

<unmentioned> → if <logic_ expr> then <stmt>

| **if** <logic _expr> **then** <matched> **else** <unmatched>.

When the same string is generated acing this unambiguous grammar, only one parse bee will be generated.

Example 3.11. Write the unambiguous BNF grammar for expressions.

Solution : The rule for expressions is that an operator of highest trinity should appear at the lowest level of the parse bee and on operator of lowest primly should appear at the highest level at the parse tree. Using this rule, the unambiguous grammar for oppressions is given as

<expr> → <expr> + <term>

| <expr> – <term>

|<term>

<term> → <term> * <factor>

|<term> | <factor>

|<factor>

<factor> → (<expr>)

|<id>

<id> → A|B|C.

The above grammar correctly reflects the rules of operator precedence and sociavity of operators.

3.3.2 CONTEXT FREE GRAMMARS (CFG)

There are well developed tools for the description of the syntax of programming languages. At about the same time, the BNF grammar was developed, a similar grammatical form, the context free grammar was developed by linguist Norm Chomsky for the definition of natural language syntax. Grammars are rewriting rules and may be used for both recognition and generation of programs. Grammars are independent of computational models and are useful for the description of the structure of languages in general. The BNF and CFG forms are equivalent in power; the difference is only in notation. For this reason the terms BNF grammar and context free grammar (CFG) are usually interchangeable in discussion of syntax.

A context free grammar (G) like BNF grammar is defined as

G = (V,T,S,P)

where V = set of Variables, non terminals or syntactic categories

T = set of terminals

S = start symbol

P = set of rules called productions.

However, the CFG grammar slightly differs from BNF grammar in terms of notations i.e

1. Unlike BNF grammars, the abstractors (non–terminals) are not enclosed in triangular brackets (< >)
2. All non-terminals or variables or syntactic categories are represented by capital case letters e.g. A, B, C, D, S etc.
3. The terminal symbols are written without any special marking like in BNF and are usually denoted by lowercase letters, digits and bold phase strings.
4. The capital letters like (X, Y, Z,) denotes the grammar symbols i.e. either terminals or non-terminals .
5. The lowercase letters u,v,w,x,y,z denotes the string of terminals.
6. The lowercase greek letters α,β,γ denotes the string of variables and terminals.
7. Unlike :: = symbol in BNF, the CFG make use of arrow (→) symbol.
8. The productions are the rules relating variables (non-terminals). The general form of a production or rule is

$A \rightarrow \alpha$

Where A is a no terminal and α is a string of terminals and non-terminal i.e. $\alpha \in (VUT)^*$

9. The production rules of CFG are called context free because the LHS of the predation can be replaced with corresponding R H S independent of context i.e. regardless of the symbol which immediately precedes or follows the non terminal on the LHS of the productions
10. A language said to be context free if and only if it can be defined in the form of context free set of productions.

Note :

1. The context fee grammars describe how lexical units (tokens) are grouped into meaningful structures.
2. The concept of derivation (LMD and RMD) parse trees and ambiguity are same as for BNF grammar.
3. The context free grammars cannot specify context–sensitive aspacts of a language such as the requirements, that a name must be declared before it is referenced, the order and number of actual parameters in a procedure cell must match the order and number of formal arguments in a procedure declaration, and that types must be compatible on both sides of an assignment operator.

Example 3.12 Consider the following context free grammar :

S→ aAS|a

A→ SbA|SS|ba.

show a parse tree and corresponding LMD for a string aabbaa.

Solution : The LMD is given as n

S ⇒ aAs
⇒ absA S
⇒ aabA S
⇒ aabbaS
⇒ aabbaa

The nonexpanding parse tree is shown in fig 3.16

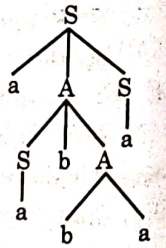


Fig. 3.16. A parse table for the string *aabbaa*

Example 3.13 :Consider the following context free grammar.

E → E +E | E* E |(E)|id.

Cheek the ambiguity of the above CFG for

(a) id + id* id

(b) id + id +id.

Solution : (a) id + id* id

The LMD is given as

E ⇒ E * E
⇒ E +E* E
⇒ id+ E * E
⇒ id+ id* E
⇒ id+ id* id

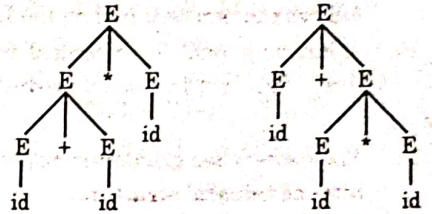


Fig. 3.17. Two distinct parse trees for the string id + id * id

There exists another LMD for the same string

E ⇒ E +E
⇒ id +E
⇒ id +E* E
⇒ id +id * E
⇒ id+ id* id.

The corresponding parse trees are shown in fig. 3.17

As there exists two distinct parse tree for the same string. Hence the grammar ambiguous.

(b) id +id+ id

The LMD is given as

E ⇒ E +E
⇒ E +E+ E
⇒ id+ E + E
⇒ id+ id+ E
⇒ id+ id+ id

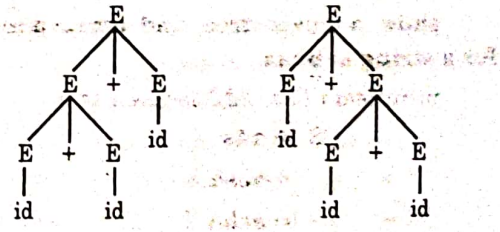


Fig. 3.18. Two distinct parse trees for the string id + id + id

here exists another LMD for the same strong.

E ⇒ E +E
⇒ id+ E
⇒ id+ E + E

⇒ id+ id+ E

⇒ id+ id+ id

As there exists two distinct parse trees for the same sing. Hence the gramm is ambiguous

Example 3.14 : Write a CFG for the expression

Solution : Using the concept of example 3.11, the unambiguous CFG for t expressions is given as

E → E + T | E – T | T

T → T* F | T/F | F

F → (E) | id

id → A| B|C

The above CFG correctly reflects the rules of operator precedence a associavity of operators.

Example 3.15 :Prove that the following BNF grammar is ambiguous.

<S> → <A>

<A> → <A> + <A> |<id>

<id> → a|b|c.

Solution : Consider a sting a+b+c

The LMD is given as

<S> → <A>

⇒ <A> + <A>

⇒ <A> + <A> + <A>

⇒ <id> + <A> + <A>

⇒ a + <A> + <A>

⇒ a + <id> + <A>

⇒ a + b + <A>

⇒ a + b + <id>

⇒ a + b + c.

There exists another LMD for the same string

<S> → <A>

⇒ <A> + <A>

⇒ <id> + <A>

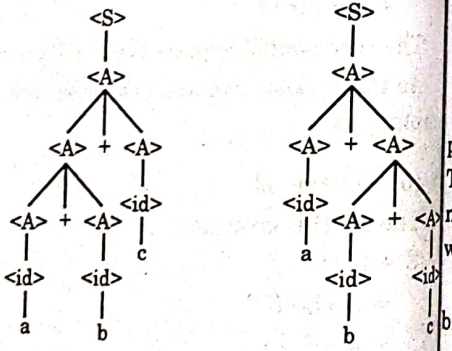


Fig. 3.19. Two distinct parse trees fo the string a + b + c

⇒ a + <A>

⇒ a + <A> + <A>

⇒ a + <id> + <A>

⇒ a + b + <A>

⇒ a + b+ <id>

⇒ a + b + c.

The a responding parse tree are shown in fig. 3.19

As there exist two distinct parse trees for the same string. Hence, the given grammar is ambiguous.

3.3.3 EXTENDED BNF (EBNF)

Despite the power, elegance, and simplicity of BNF grammars, they are not an ideal notation for communicating the rules of programming language syntax to the practicing programmer. The primary reason is that the simplicity of the BNF rule forces a rather unnatural representation for the common syntactic constructs of optional elements, alternative elements and repeat·i elements within a grammar rule. For example, to express the simple syntactic idea **"a signed integer is a sequence of digits preceded by an optional plus or minus."** The complex set of recursive rules using BNF are given as

<signed integer> :: = + <integer> | – <integer>

<integer> :: = <digit> | <integer> <digit>

In order to avoid some of these unnatural ways to specify simple syntactic properties, three extensions are commonly included in the various versions of BNF. These extended versions of BNF are celled extended BNF (EBNF). The extensions do not enhance the descriptive power of BNF; they only increase its readability and writability. The following three extensions are included in EBNF

1. An optional element may be indicated by enclosing the element in square brackets [...]. For example the if- then- else statement in C can be described as

BNF <selection> →if (<expr>) <stmt>;

| if (<expr>) <stmt> else <stmt>;

EBNF : <selection>→ if (<expr>) <stmt> [else <stmt>];

<selection>

2. A choice of alternatives may use the '|' symbol within a single rule, optionally enclosed by parenthesis if needed. For example consider the for statement in Pascal:

BNF : <for_ stmt>→ for <var> : =<expr> to <expr> do <stmt>
| for <var> : = <expr> down to <expr> do <stmt>

EBNF : <for _ stmt> → for <var> : = <expr> (to| down to) <expr> do <stmt>

3. An arbitrary sequence of instances of an element may be indicated b enclosing the element in braces followed by an as trick {...}*. For example, conside the list of identifiers.

BNF : <ident _ list>→<identifier>
< identifier >, <ident _list>

EBNF : <ident _ list> → < identifier > {,< identifier >}*

Example 3.16. The BNF grammar for a signed integer is given as

<signed integer > ::= + <integer> | – < integer >

<integer> ::= <digit> | <integer> <digit>

Write the EBNF for the above BNF grammar.

Solution :<signed integer> ::= [+|–] <digit> {<digit>}*

Example 3.17. Write the EBNF version of an expression grammar.

Solution : The BNF grammar for an previsions is given as

BNF : <expr> →<expr> + <term>
| <expr> –<term>
| <term>
<term> → <term> × <factor>
| <term>/<factor>
| <factor>
<factor> → (<expr>)
| <id>
<id> → A|B|C.

The corresponding EBNF for expressing is given as

EBNF : <expr> → <term> {(+ | –) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → (<expr>)
|<id>.

3.3.4. SYNTAX GRAPHS

A graph G is defined as a collection of nodes (or vertices) connected by lines called edges. Formally is defined as a graph G = (V,E) where

V = Set of vertices

E = Set of edges.

A graph is said to be **directed** if the direction of the edges is shown explicitly otherwise it is called an **undirected** graph.

The syntax graphs (also called **syntax charts** and **syntax diagrams**) are used to represent the information contained in BNF and EBNF, graphically. The following conventions are used for drawing syntax graphs. The syntax increases graph the readability by allowing as to visualize it in 2D.

1. Each syntactic unit (e.g. a nonterminal) is represented by a separate syntax graph.
2. The syntactic units (nonterminals) are represented using rectangular nodes.
3. The terminal symbols are represented using circles or ellipses.

As an example to illustrate the concept of syntax graphs consider the following BNF = grammar for Pascal if–then–else statement. The BNF, EBNF and syntax graph are shown in fig. 3.20

BNF : <if_stmt> → if <logic_ expr> then <stmt> ;
| if <logic _expr> then <stmt> else <stmt>;

EBNF : <if _ stmt> → if <logic _expr> then <stmt> [else <stmt>];

Syntax graph :

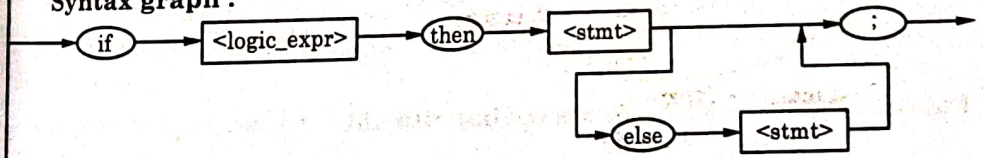


Fig. 3.20. BNF, EBNF and Syntex graph for Pascal **if-then-else** statement

3.3.5 FINITE STATE AUTOMATA

Lexemes are the smallest possible units of a program. A token of a language is a category of its lexemes. The examples of tokens includes identifiers, constants, literals, operators, keywords and punctuation symbols. A simple model that recognizes such tokens is called a finite state automata (or simply finite automata).

A finite state automata has a starting state, one or more final states, and a of transitions (labeled arcs) from one state to another. Any string that takes machine from the initial state to a final state through a series of transitions accepted by the machine. Mathematically, a finite state automata is defined as

$M = (Q, \Sigma, \delta, q_0, F)$

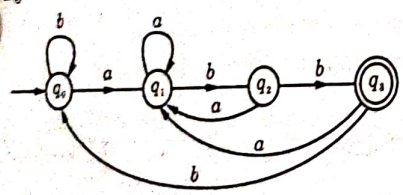
Where Q = Set of states
Σ = Input alphabet
δ = Transition function
q_0 = Initial State
F = Set of Final or Accepting states.

A Finite state automata is called

1. **Deterministic Finite State Automata (DFA) :–** If for each state of finite state automata and for each input symbol there is exactly one transition to same or a different state, then the corresponding finite state automata is c **Deterministic Finite Automata (DFA).** The transition function for a DAF is g as

$$\delta : Q \times \Sigma \to Q$$

For example, Consider a DFA for all the strings ending with abb as show figure 3.21. Here $Q = \{q_0, q_1, q_2, q_3\}, \Sigma = \{a,b\}, q_0 = q_0, F = \{q_3\}$



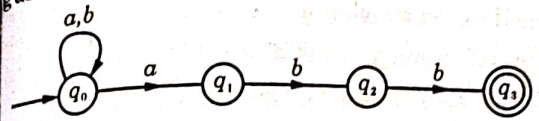
$\delta(q_0,a) = q_1$, $\delta(q_0,b) = q_0$
$\delta(q_1,a) = q_1$, $\delta(q_1,b) = q_2$
$\delta(q_2,a) = q_1$, $\delta(q_2,b) = q_3$
$\delta(q_3,a) = q_1$, $\delta(q_3,b) = q_0$

Fig. 3.21. A DFA for all the strings ending with abb

2. **Nondeterministic Finite Automata (NFA) :** If for each state of finite state automata, and for each input symbol there are zero, one or m transitions to the same or different states, then the corresponding finite s automata is called **non-deterministic finite automata (NFA).** The transi function for a NFA is given as

$$\delta : Q \times \Sigma \to 2^{|Q|}$$

For example consider a NFA for all the strings ending with abb as shown in gure 3.22; Here $Q = \{q_0, q_1, q_2, q_3\}, \Sigma = \{a,b\}, q_0 = q_0, F = \{q_3\}$



$\delta(q_0,a) = \{q_0, q_1\}$ $\delta(q_0,b) = q_0$,
$\delta(q_1,a) = \phi$ $\delta(q_1,b) = q_2$,
$\delta(q_2,a) = \phi$ $\delta(q_2,b) = q_3$,
$\delta(q_3,a) = \phi$ $\delta(q_3,b) = \phi$.

g. 3.22. A NFA for all the strings **ending with abb**

A further flexibility can be provided in NFA by allowing null ($\wedge$) moves. In that se NFA is called NFA with $\wedge$ moves.

3.6 REGULAR GRAMMARS AND REGULAR EXPRESSIONS

Regular grammars are special cases of context free grammar and is equivalent the finite state automata languages as described in section 3.3.5. A grammar is id to be regular if all its productions are either in left linear or Right linear from.

Left Linear Grammar : A grammar G = (V, T, S, P) is said to be left linear if its productions are of the form

$$A \to Bw$$
$$A \to w$$

Where A and B are non terminals and w is a string of terminals

Right Linear Grammar : A grammar G = (V, T, S, P) is said to be right linear if

$$A \to W\,B$$
$$A \to w$$

Where A and B are non terminals and w is a strings of terminals. For example regular grammar for generating a binary string ending with 0 can be described following rules.

Left Linear Grammar :

$$S \to A\,0$$
$$A \to A0 \mid A1 \mid \wedge$$

Right Linear Grammar :

$$S \to 0S \mid 1S \mid 0$$

6.1 REGULAR EXPRESSIONS

While CFGs can be used to describe the tokens of a programming language, ular expressions are a more convenient notation for describing the structure of

tokens. Regular expressions are mathematical expressions used to describe tok
The regular expressions can be defined recursively as

1. The regular expressing for the ϕ language is ϕ.
2. The regular expressing for the language $\{\wedge\}$ is $\wedge$.
3. The regular expressing for a terminal symbol a is a.
4. If r and s are the regular expressions for the language L_R and L_S then
   (a) r + s (or r | s) is a regular expressing for L_R U L_S
   (b) rs is a regular expressing for L_R . L_S
   (c) r^* is a regular expressing for $(L_R)^*$
   Here '|' or '+' is used to separate alternatives, '.' to denote concatena
   and, '*' to indicate that the previous item may be repeated zero or
   times.
5. Nothing else is a regular expression.

As an example of regular expressions consider the description of the t
identifier.

identifier = letter (letter+ digit)*

3.3.7 PUSH DOWN AUTOMATA (PDA)

A push down automata (PDA) is an abstract model machine similar to finite state automata. It also has a finite set of states. However, in addition, it h pushdown stack mathematically a pushdown automata is defined as

$$M = (Q, \Sigma, \delta, q_0, \Gamma, Z_0, F)$$

Where Q = Set of states.
Σ = Input Alphabet
δ = Transition function
q_0 = Initial state
Γ = Stack Alphabet.
Z_0 = Initial Stack symbol
F = Set of final state

The transition of a pushdown automata in a particular state involves rea the current input symbol and a top of stack, entering a new state and updating stack top i.e. the transition function of a PDA is given as

$$\delta : Q \times (\Sigma \cup \{\wedge\}) \times \Gamma \rightarrow Q \times \Gamma^*$$

If after the processing of input string either the stack is empty or PDA is in o of final states, the string is said to be accepted. Further details of pushdow Automata are beyond the scope of this book.

3.4 ATTRIBUTE GRAMMAR

There are some characteristics of the structure of programming languages that are difficult to describe with BNF, and some that are impossible. The type compatibility rules are difficult to implement using BNF and the rule that all variables must be declared before they are referrenced can not be specified using BNF. These difficulties exemplify the concept of static semantic rules. The static semantics of a language is only indirectly related to the meaning of program during execution. The analysis required to check the specifications can be done at compile time. Hence static semantics are so named. The static semantic rules of a language state its type constraints. The static semantics are described using more powerful mechanisms called **attribute grammar.**

The idea behind attribute grammars is to associate a function with each node in the parse tree of a program giving the semantic content of that node. Attribute grammars are created by adding **attributes, attribute computation functions and predicate functions.**

1. **Attributes :** Attributes refer to some well defined characteristics of grammar symbols and are generally similar to variables with values assigned to them. For a grammar symbol X the associated set of attributes A(X) consists of two disjoint sets S (X) and I (X), called syntnesised and inherited attributes, as shown in

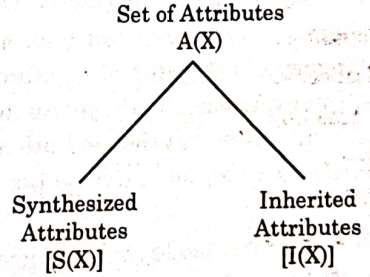


Fig. 3.23. Synthesized and Inherited Attributes

(a) **Synthesized Attributes :** A synthesized attribute is a function that relates the LHS nonterminal to the values of the RHS nonterminal. These attributes pass information up a parse tree i.e. were "synthesized" from the information below in the tree.

(b) **Inherited Attributes :** An inherited attribute is a function that relates the nonterminal values in a tree with nonterminal values higher up in the tree. These attributes pass information down a parse tree.

2. **Attribute Computation Functions :** Attribute computation functions (or semantic rules) are associated with grammar rules to specific how the attribute values are computed. We define the attribute computation functions for synthesized and inherited attributes separately.

(a) **Attribute computation functions for synthesized attributes :** For a grammar rule $X_0 \rightarrow X_1\ X_2 \ldots\ldots X_n$, the synthesized attributes of X_0 are computed by using an attribute Computation function of the form

$$S(X_0) = f(A(X_1), \ldots\ldots, A(X_n))$$

i.e. the value of a synthesized attribute on a parse tree depends only on the values of the attributes on that node's children nodes e.g. consider a rule : $A \rightarrow XYX$. The associated semantic rule for synthesized attribute A can be A. Val = X. Val + Y. Val + Z. Val

(b) ***Attribute Computation functions for Inherited attribute :*** The attribute computation functions for the nonterminals on the RHS of any rule are defined as a function of the LHS nonterminal. For a grammar rule $X_0 \rightarrow X_1\ X_2 \ldots\ldots X_n$ the inherited attributes of symbols X_j $(1 \le j \le n)$ are computed using an attribute computation function of the form

$$I(X_j) = f(A(X_0), \ldots A(X_n)).$$

i.e. the value or an inherited attribute on a parse tree depends on the attribute values of that node's parent node and those of its sibling nodes.

3. **Predicate Functions :** The predicate functions states some of the syntax and semantic rules of the language, associated with grammar rules. A predicate function is in the form of boolean expression on the attribute set $\{A(X_0) \ldots, A(X_n)\}$. A true value of predicate function indicates that the derivation is allowed while a false predicate function value indicates a violation of the syntax or semantic rules of the language.

Now we discuss two more terms related to attribute grammars.

1. **Intrinsic attributes :** These are the synthesized attributes of leaf nodes whose values are determined outside the parse tree e.g. the type of an instance of variable could come from symbol table (A data structure used to store variable names and their types).

2. **Fully attributed parse tree :** A parse tree is said to be fully attributed parse tree if all the attribute values in a parse tree have been computed.

Now we discuss an example of attribute grammar. Consider a BNF grammar for assignment statement

```
<assign> → <var> : = <expr>
<expr>   → <var> + <var>
         | <var>
<var>    → A | B | C.
```

We associate two types of attributes with grammar symbols i.e.

(a) **actual_type :–** A synthesized attribute associated with <var> and <expr> and is used to store actual type (int or real).

(b) **expected_type :–** An inherited attribute associated with nonterminal <expr> and is used to store expected type for the expression (int or real) as determined by the type of the variable on LHS of the assignment statement. The attribute grammar may be written as :

Syntax rule	Semantic rule	Predicate function
1. <assign> → <var> : = <expr>	<expr>. expected_type ← <var>. actual_type	—
2. <expr> → <var> [2] +<var> [3]	<expr> actual_type ← if (<var> [2]. actual_type = int) and (<var> [3]. actual_type = int) then int else real end if	<expr>.actual_type= <expr>.expected_type.
3. <expr> → <var>	<expr>. actual_ type ← <var>. actual_type	<expr>. actual_type = <expr> . expected _type
4. <var> → A | B | C	<var>.actual_type←look_up<var>.string)	—

Fig. 3.8.4 : Attribute grammar for simple assignment statement.

The attributes are computed in the following order.

1. <var> . actual_type ← look_up (A) [Rule 4]
2. <expr>. expecte_type ← <var>. actual_type [Rule1]
3. <var> [2]. actual_type ← look_up (A) [Rule 4]
   <var> [3]. actual_type ← look _up (B) [Rule 4]

4. <expr>. a actual_type ← either int or real [Rulel 2]
5. <expr>. expected_type = <expr>. actual_type is either TRUE or FALS [Rule 2]

The flow of attributes is shown in figure 3.24.

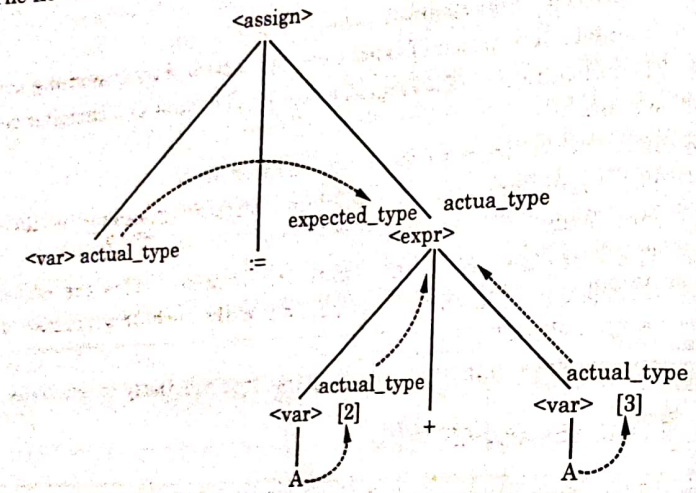


Fig. 3.24. The flow of attributes in the parse tree

If we consider A to be defined as real and B as an integer then the full attributed parse tree is as shown in figure 3.25

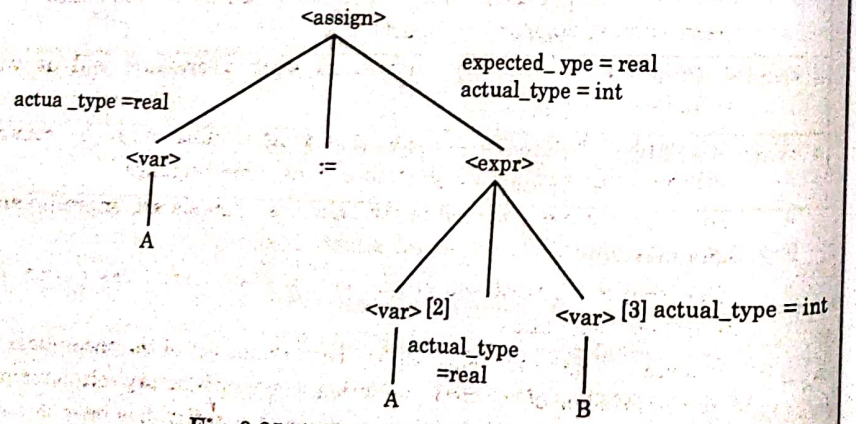


Fig. 3.25. A fully attrubuted parse tree

3.4.1 APPLICATIONS AND DIFFICULTIES

The various applications of attribute grammar are as :

1. Natural language processing system.
2. Syntax directed editing system
3. To describe type capability rules.
4. Complete description of syntax and semantics of programming languages,
5. Formal definition of a language that can be input to a compiler generation system.

The difficulties in using an attribute grammar are as.

1. The main difficulty in using an attribute grammar is its size and complexity.
2. The large number of attributes and semantic rules are required for a complete programming language that make such programmers difficult to write and read.
3. The attribute values an a large parse tree are costly to evaluate.

3.5 DYNAMIC SEMANTICS

As discussed earlier the syntax of a programming language is the form of its expressions, statements and program units. The semantics (or dynamic semantics) describes the meaning, of the expressions, statements, and program units. There are several reasons for the description standardization of semantics which are given as :

1 The different readers may interprete different semantics of a programming language construct.
2. A programmer may misunderstand what a program will do when executed.
3. An implementer may implement a programming language construct differently from other implementors of the same language.
4. The explanations provided in the language manuals are imprecise and in complete.
5. The program correctness proofs rely an some formal description of the language semantics.

The problem of semantic definition has been the object of the theoretical study for as long as the problem of syntactic definition, but a satisfactory solution has been much more difficult to find. No universally accepted notation has been devised for

dynamic semantics. Many different methods for the formal definition of semantics has been developed. The following are some of these.

3.5.1. OPERATIONAL SEMANTICS

Operational semantics are used when learning a programming language and by compiler writer. The operational semantics were first used to describe the semantics of PL/I. The particular abstract machine and the translation rules for PC/I were together named **Vienna Definition language (VDL)**, after the city where it was devised by IBM. An operational definition of a programming language is a definition that defines how programs in the language are executed on machine either real or simulated. Operational semantics may describe the syntactic transformations which mimic the execution of the program on an abstract machine or define a translation of the program into recursive functions. This means that operational semantics describe the meaning of a program by executing its statements on a machine. The changes occurring in machine's state describe the meaning of the statement (a state being the value of registers memory locations).

An operational definition of a language L consists of two parts:

1. **A translator,** to convert the statement of Language L to the chosen low level language.
2. **The virtual machine for that low–level language.** The state changes in the virtual machine brought about by executing the code that results from translating a given statement in the high–level language defines the meaning of that statement.

As an example of the operational semantics, consider the C **for** construct that can be described in terms of very simple instructions as in

C statement	operation semantics
for (expr1; expr2; expr3)	expr1;
{	loop : if expr2 = 0 go to out
.....	.....
}	expr3;
	goto loop
	out :.....

The human reader of such a description is the virtual computer and is assumed able to correctly "execute" the instructions in the definition and recognize the s of the "execution". In the end we give following remarks regarding operational tics.

1. Operational semantics provide an effective means of describing semantics for language users and language implementers as long as the descriptions are kept simple and informal.
2. operational semantics depends on algorithms, not mathematics.

3.5.2 DENOTATIONAL SEMANTICS

The concept of denotational semantics is based on mathematical objects and recursive functions. The mathematical objects are used to represent the meanings of language constructs. The language entities are converted to these mathematical objects with recursive functions. The method is named **denotational** because the mathematical objects denote the meaning of their corresponding syntactic entities. The key difference between operational semantics and denotational semantics is that the state changes in operational semantics are defined by coded algorithms, whereas in denotational semantics, state changes are defined by rigorous mathematical functions. As an example of denotational semantics consider the very simple language construct, decimal numbers. The syntax of decimal numbers can be described by the following grammar rules

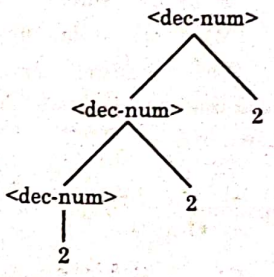


Fig. 3.26. A parse tree for the decimal number 222.

<dec_num> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

| <dec _num> (0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)

The parse for the example decimal number 222 is shown in Figure 3.26

Now we define a semantic function M_{dec} for mapping the syntactic objects (decimal numbers) as:

M_{dec} ('0') = 0
M_{dec} ('1') = 1
M_{dec} ('2') = 2
M_{dec} ('3') = 3
M_{dec} ('4') = 4
M_{dec} ('5') = 5
M_{dec} ('6') = 6
M_{dec} ('7') = 7
M_{dec} ('8') = 8
M_{dec} ('9') = 9

M_{dec} (<dec_ num> '0') = 10* M_{dec} (<dec_ num>)
M_{dec} (<dec_ num> '1') = 10* M_{dec} (<dec_ num>) +1
M_{dec} (<dec_ num> '2') = 10* M_{dec} (<dec_ num>) +2
M_{dec} (<dec_ num> '3') = 10* M_{dec} (<dec_ num>) +3
M_{dec} (<dec_ num> '4') = 10* M_{dec} (<dec_ num>) +4
M_{dec} (<dec_ num> '5') = 10* M_{dec} (<dec_ num>) +5
M_{dec} (<dec_ num> '6') = 10* M_{dec} (<dec_ num>) +6
M_{dec} (<dec_ num> '7') = 10* M_{dec} (<dec_ num>) +7
M_{dec} (<dec_ num> '8') = 10* M_{dec} (<dec_ num>) +8
M_{dec} (<dec_ num> '9') = 10* M_{dec} (<dec_ num>) +9

The resulting parse tree after attaching the meanings or denoted objects (decimal number) is shown in figure 3.27.

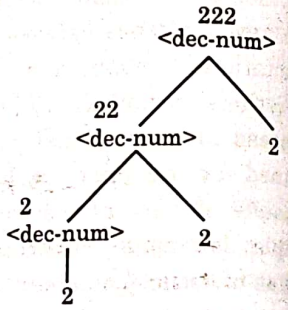


Fig. 3.27. A parse tree for denoted objects for 222.

In the end we give the following remarks regarding denotational semantics.

1. The denotational semantics are syntax–directed semantics based on the concept of mathematical objects and recursive functions.
2. The denotational semantics provide a framework for thinking about programming in a highly rigorous way.
3. The denotational semantics can be used as an aid to language design.
4. The denotational semantics are used in the theoretical and comparative studies of programming languages.
5. The denotational Semantics are used for automatic construction of compilers for the programming languages.
6. The main difficulty with the denotational semantics live in creating the objects and the mapping functions.
7. The denotational semantics do not provide much guidance to the implementer.
8. The denotational semantics are too complex to be of direct value to users.
9. The denotations other than mathematical objects are possible .e.g. compiler writer would prefer, that the object denoted would be appropriate object code.

Example 3.18 : Explain the concept of denotational semantics of binary numbers in a programming language.

Solution : The syntax of binary numbers can be described by the following grammar rules :

<bin_num> →0 | 1
| <bin_num> 0 | <bin_num>1

A parse tree for the example binary number, 100 is shown in figure 3.28

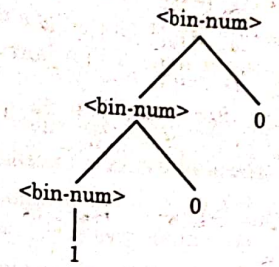


Fig. 3.28. A parse tree for a binary number 100.

Now we define a semantic function M_{bin} for mapping the syntactic objects (binary numbers) as :

M_{bin} ('0') = 0
M_{bin} ('1') = 1
M_{bin} (<bin_num>'0') = 2* M_{bin} (<bin_num>)
M_{bin} (<bin_num>'1') = 2* M_{bin} (<bin_num>) +1

The resulting parse tree after attaching the meanings or denoted objects (binary numbers) is shown in Figure 3.29

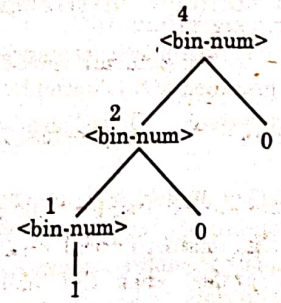


Fig. 3.29. A parse tree with denoted objects for 100.

3.5.3. AXIOMATIC SEMANTICS

Axiomatic semantics are based an formal and mathematical logic and ar devised as a tool for proving the correctness of programs (correctness of a progra means that the program performs the computation described by its specificatio The axiomatic semantics defines the meaning of the program explicitly. Th axiomatic semantics of a programming language are the assertions (the logic expressions also called predicates) about relationships that remain the same eac time the program executes. Axiomatic definitions are defined for each contr structure and command. The axiomatic semantics of a programming language defi a mathematical theory of programs written in a language. A mathematical theor has three components :

1. **Syntactic rules :** The syntactic rules determine the structure formulas which are the statements of interest.
2. **Axioms :** An axiom is a logic statement that is assumed to be true. Thes axioms describe the basic properties of the system.
3. **Inference rules :** These are the mechanisms for deducing (sometime called inferring) new theorems from axioms and other theorems.

The semantic formulas are triples of the form.

$$\boxed{\{P\}\ S\ \{Q\}}$$

Where S is a statement (a command or control structure) in th programming language, P and Q are assertions (logical expressions) o statements concerning the properties of program objects (often progra variables) which may be true or false. P is called a **precondition** and Q called a **post–condition.** These terms are discussed in detail in nex section.

3.5.3.1 PRECONDITIONS, POST CONDITIONS AND WEAKEST PRECONDITIONS

From the semantic formula for axiomatic semantics, it is clear that eac statement S of a program is both preceded and followed by a logical expression tha species the constraints on program variables. These logical expression (assertion) ar called.

1. **Precondition :** An assertion immediately preceding a program statemen S that describes the constraints on program variables at that point called a precondition. In the axiomatic semantics of the form {P} S {Q}, {P is used to denote a precondition.

2. **Post Condition :** An assertion immediately following a program statement S that do describes new constraints on program variables (and possibly others) after the execution of the statement is called a post condition. In the axiomatic semantics of the form {P} S {Q}, {Q} is used to denote a post condition.

To illustrate the concept of precondition and postcondition described above consider the following semantic notation.

Sum = 2* x + 1; {sum> 1}

Here {sum> 1} represents a post condition. One possible precondition satisfying the given constraints is {x >10} i.e. we may write

{x > 10} sum = 2*x + 1; {sum >1}

3. **Weakest Precondition:** The weakest precondition is defined as the least restrictive precondition that will guarantee the validity of the associated post condition. For example in the semantic notation

sum = 2*x + 1; {sum >1}

All preconditions like {x > 10}, {x > 50} and {x >100} are valid preconditions but the weakest of all preconditions that guarantee the validity of the associated position is {x > 0 } in this case.

3.5.3.2 THE PROGRAM CORRECTNESS PROOFS AND WEAKEST PRECONDITIONS

The Axiomatic semantics is a powerful tool for research into program correctness proofs. The program correctness proofs can be constructed by computing the weakest preconditions. The computation of a weakest precondition from the statement and a postcondition is simple and can be specified by an axiom. In most cases, the weakest postcondition can be computed only by an inference rule. Therefore, we describe the concept of computing the weakest preconditions in different programming language constructs.

1. **Assignment Statements :** consider an assignment statement of the form.

$$\boxed{x = E}$$

The corresponding preconditions and postcondition are represented by P and Q

i.e. $\boxed{\{P\}\ x = E\ \{Q\}}$ represent the axiomatic description of an assignment statement. The precondition {P} can be computed as Q with all instances of x replaced by E. i.e.

$$\boxed{P = Q_{x \to E}}$$

Therefore, the resultant axiomatic notation becomes

$$\boxed{\{Q_{x \to E}\} x = E \{Q\}}$$

Example 3.19. Compute the weakest precondition for each of the following assignment statements and post conditions :

(a) $a := 2 * (b - 1) - 1 \{a > 0\}$

(b) $b := (c + 10) \mid 3 \{b > 6\}$

(c) $x := 2 * y - 3 \{x > 25\}$

(d) $a := a + 2 * b - 1 \{a > 1\}$

(e) $a := b/2 - 1 \{a < 10\}$

(f) $x := 2 * y + x - 1 \{x > 11\}$

(g) $x := x + y - 3 \{x > 10\}$

Solution : (a) $a := 2 * (b-1) - 1 \{a > 0\}$

The weakest precondition is computed by substituting $2 * (b-1) - 1$ in the assertion $\{a > 0\}$, as follows:

$2 * (b-1) - 1 > 0$

$2 * (b-1) > 1$

$(b-1) > 1/2$

$b > 1/2 + 1$

$$\boxed{b > \frac{3}{2}}$$

Thus, the weakest precondition for the given assignment and postcondition $\{b < 22\}$.

(b) $b := (c + 10) \mid 3 \{b > 6\}$.

The weakest precondition is computed by substituting $(c + 10)/3$ in the assertion $\{b > 6\}$, as follows :

$(c + 10)/3 > 6$

$(c + 10) > 18$

$$\boxed{c > 8}$$

Thus, the weakest precondition for a given statement and postposition is $\{c > 8\}$.

(c) $x := 2 * y - 3 \{x > 25\}$

The weakest precondition is computed by substituting $2 * y - 3$ in the assertion $\{x > 25\}$, as follows:

$2 * y - 3 > 25$

$2 * y > 28$

$$\boxed{y > 14}$$

Thus, the weakest precondition for a given statement and postposition is $\{y > 14\}$.

(d) $a := a + 2 * b - 1 \{a > 1\}$

The weakest precondition is computed by substituting $a + 2 * b - 1$ in the assertion $\{a > 1\}$, as follows:

$a + 2 * b - 1 > 1$

$a + 2 * b > 2$

$2 * b > 2 - a$

$b > (2 - a) \mid 2$

$$\boxed{b > 1 - \frac{a}{2}}$$

The weakest precondition for a given statement and postposition is $\{b > 1 - a/2\}$

(e) $x := 2 * y + x - 1 \{x > 11\}$.

The weakest precondition is computed by substituting $2 * y + x - 1$ in the assertion as follows:

$2 * y + x - 1 > 11$

$2 * y + x > 12$

$2 * y > 12 - x$

$$\boxed{y > 6 - \frac{x}{2}}$$

Thus, the weakest precondition for a given statement a postcondition is $\{y > 6 - x/2\}$

(f) $a := b/2 - 1 \{a < 10\}$

The weakest precondition is computed by substituting $b/2 - 1$ in the assertion $\{a < 10\}$ as follows :

$b/2 - 1 < 10$

$b/2 < 11$

$\boxed{b < 22}$

Thus, the weakest precondition for a given statement and postcondition is $\{b < 22\}$.

(g) $x := x + y - 3 \{x > 10\}$

The weakest precondition is computed by substituting $x + y - 3$ in the assertion $\{x > 10\}$, as follows:

$x + y - 3 > 10$

$x + y > 13$

$\boxed{y > 13 - x}$

Thus, the weakest precondition for a given statement and postposition is $\{13 - x\}$.

1. **Rule of consequence :** The rule of consequence (Inference rule) states that if $S_1\ S_2, \ldots\ldots S_n$ are true, then the truth of S can be inferred mathematically. The general form of an inference rule is

$$\frac{S_1\ S_2 \ldots\ldots S_n}{S}$$

The form of rule of consequences is

$$\frac{\{P\}\ S\ \{Q\},\ P' \Rightarrow P, Q \Rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$$

i.e. If the logical statement $\{P\}\ S\ \{Q\}$ is true, the assertion $P' \Rightarrow P$ and the assertion $Q \Rightarrow Q'$ then it can be inferred that $\{P'\}\ S\ \{Q'\}$. For example let $P = \{x > 5\}$, Q and Q' be $\{x > 0\}$, P' be $\{x > 7\}$

$$\frac{\{x > 5\} x = x - 5 \{x > 0\} (x > 7) \Rightarrow (x > 5), (x > 0) \Rightarrow (x > 0)}{\{x > 5\} x = x - 3 \{x > 0\}}$$

2. **Sequence of Statements :** Let S_1 and S_2 be adjacent assignment statements. i.e. S_1 is $x_1 = E_1$ and S_2 is $x_2 = E_2$. If S_1 and S_2 have the following pre-and post conditions

$\{P_1'\}\ S_1 \{P_2\}$

$\{P_2\}\ S_2\ \{P_3\}$

The inference rule for such a two – statement sequence is

$$\frac{\{P_1\} S_1 \{P_2\}, \{P_2\} S_2 \{P_3\}}{\{P_1\} S_1; S_2 \{P_3\}}$$

This means that for $\{P_1\}\ S_1, S_2\ \{P_3\}$

$\{P_{3\ x_2 \to E_2}\} x_2 = E_2 \{P_3\}$

$\{(P_{3\ x_2 \to E_2}) x_1 \to E_1\} x_1 = E_1 \{P_{3\ x_2 \to E_2}\}$

and the weakest prenditions $\{(P_{3\ x_2 \to E_2})\ x_1 \to E_1\}$

Example 3.20:Compute the weakest precondition for each of the following sequences of assignment statements and their post conditions :

(a) $a := a * b + 1;$

$b := a - 3$

$\{b < 0\}.$

(b) $y := 3 * x + 1;$

$x := y + 3;$

$\{x < 10\}.$

(c) $a := 3 * (2 * b + a);$

$b := 2 * a - 1;$

$\{b > 5\}$

Solution : (a) $a := 2 * b + 1;$

$b := a - 3;$

$\{b < 0\}$

The weakest precondition for the last assignment statement is calculated

$a - 3 < 0$

$\boxed{a < 3}$

This condition is used as the post condition for the first. The precondition for the first assignment statement can now be computed as;

$2 * b + 1 < 3$

$2 * b < 2$

$\boxed{b < 1}$

(b) $y := 3 * x + 1;$

$x := y + 3;$

$\{x < 10\}$

The weakest precondition for the last assignment statement is calculated as

$y + 3 < 10$

$y < 7$

This condition is used as the postcondition for the first. The precondition for the first assignment can now be computed as

$3 * x + 1 < 7$

$\boxed{x < 2}$

(c) $a := 3 * (2 * b + a);$

$b := 2 * a - 1$

$\{b > 5\}$

The weakest precondition for the last assignment is calculated as

$2 * a - 1 > 5$

$2 * a > 6$

$\boxed{a > 3}$

This condition is used as the postcondition for the first . The precondition for the first assignment statement can now be computed as.

$3 * (2 * b + a) > 3$

$2 * b + a > 1$

$2 * b > 1 - a$

$\boxed{b > (1 - a)/2}$

3. **Selection Statements :** Consider a selection statement of the form

if B **then** S_1 **else** S_2

Let's assume that the given post conation is {Q}. The method of calculating the precondition is as

(*i*) Compute the precondition {P} when B is the i.e. the precondition of the alternate. ({Band P} S_1{Q}).

(*ii*) Compute the precondition {P} when B is false i.e. the precondition of 'else' alternate. ({(not B) and P} S_2 {Q}.

(*iii*) For the precondition calculated above in (*i*) and (*ii*) apply the rule of consequences to derive the precondition of the selection statement i.e. { P} if B then S_1 else S_2 {Q}.

Example 3.2. Compute the weakest precondition for the following selection statement and the post condition .

if $(x > 0)$

$y = y - 1$

else $y = y + 1$ $\{y > 0\}$.

Solution: For **'then'** clause

$y = y - 1\{y > 0\}$

$y - 1 > 0$

$\Rightarrow \boxed{y > 1}$

For **'else'** clause ...(*i*)

$y = y + 1$ $\{y > 0\}$

$y + 1 > 0$

$\Rightarrow \boxed{y > -1}$

From (*i*) and (*ii*) we apply the rule of consequence i.e. ...(*ii*)

$\{y > 1\} \Rightarrow \{y > -1\}$.

Therefore the weakest precondition of the selection statement is $\{y > 1\}$

4. **Logical Pretest Loops :** Consider a **while** loop

while B **do** S **end.**

The axiomatic description of a **while** loop is written as

{P} while B do S end {Q}.

The inference rule for computing the precondition for **a while** loop is

$$\frac{\text{(I and B) S \{I\}}}{\text{\{I\} \textbf{while} B \textbf{do} S \textbf{end} \{I and (not B)\}}}$$

Here, the assertion I is celled **loop invariant** and the determination of I is crucial to finding the weakest precondition. After the loop invariant is determined, the complete axiomatic description of a **while** construct requires all of the following to be true, in which I is the loop invariant.

(*i*) $P \Rightarrow I$

(*ii*) {I} B {I}

(*iii*) {I *and* B} S {I}

(*iv*) (I *and* (*not* B)) $\Rightarrow$ Q

(*v*) the loop terminates.

The important step however remains the computation of loop invariant. The loop invariant can be computed similar to induction hypothesis in mathematical induction. After computing the relationship for a few cases, the generalization is done. The process of producing a weakest precondition as a function *wp* i.e.

wp (statement, postcondition)= precondition

Example 3.22 Compute the weakest precondition for the follow 'while' loop and the post condition.

while $y <> x$ *do* $y = y + 1$ *end* $\{y = x\}$

Solution: First we determine the loop invariant I. For that we compute t relationship for few iteration

For **zero** iterations, the weakest precondition is $\{y = x\}$

For **one** iteration, the weakest precondition is computed as

$wp\ (y = y + 1, \{y = x\}) = \{y + 1 = x\}$ *or* $\{y = x - 1\}$

For **two** iteration, the weakest precondition is computed as

$wp\ (y = y + 1, \{y = x - 1\}) = \{y + 1 = x - 1\}$ *or* $\{y = x - 2\}$.

For **three** iteration it is

$wp\ (y = y + 1, \{y = x - 2\}) = \{y + 1 = x - 2\}$ *or* $\{y = x - 3\}$.

When we generalize the results of these iterations, we get the loop invariant

$\boxed{I = \{y <= x\}}$

Also, Here P = I can be used.

Now we will verify the criteria (*i*) through (*v*) for our example loop.

(*i*) As P = I, clearly P⇒ I.

(*ii*) The I is unaffected by the evaluation of the boolean loop expression, wh is $y <> x$. This expression changes nothing, so it cannot affect I.

(*iii*) {I *and* B} S {I}

$\{y <= x \text{ and } y <> x\}\ y = y + 1\ \{y <> x\}$

Proof: Using $y = y + 1\ \{y <= x\}$

$y + 1 <= x$

$y <= x - 1$,which is equivalent to $\{y <= x \text{ and } y <> x\}$. Hence prov

(*iv*) (I *and* (*not* B)) ⇒ Q

$\{(y <= x) \text{ and not } (y <> x)\} \Rightarrow \{y = x\}$

$\{(y <= x) \text{ and } (y = x)\} \Rightarrow \{y = x\}$

$(y = x)\} \Rightarrow \{y = x\}$. Hence proved.

(*v*) Here we consider then loop termination. The question is whether

$\{y <= x\}$ **while** $y <> x$ **do** y n= $y + 1\ \{y = x\}$ terminates. Assuming x a to be of integer type, we can easily see that the loop terminates. The precondit guarantees that y initially is not grater than x . The loop body increases y in e

iteration unstill $y = x$. No matter how much smaller y is than x initially, it will eventually become equal to x. So, the loop will terminate.

Because our choice of I satisfies all the five criteria, it is adequate for the loop invariant and loop precondition.

3.5.3.2.1 THE PROGRAM CORRECTNESS PROOFS

The correctness proofs can be constructed for programs by using the following steps.

1. Use the last statement of the program and given post condition to compute its weakest precondition and then working backwards through the program, computing weakest preconditions for each statement until the beginning of the program is reached. The first precondition states the conditions under which the program will compute the desired results.
2. If the computed weakest precondition and the given weakest preconditions for the first statement are same then the give program is correct else is incorrect.

Example 3. 23. Prove the following program is correct :

$\{x = V_x \text{ and } y = V_y\}$

temp = *x* ;

***x* = *y* ;**

***y* = temp :**

$\{x = V_y \text{ and } y = V_x\}$

Solution : For the lest statement

$wp\ (y = temp; \{x = V_y \text{ and } y = V_x\}) = \{x = V_y \text{ and } temp = V_x\}$

Next continuing backwards we get.

$wp\ (x = y; \{x = V_y \text{ and } temp = V_x\}) = \{y = V_y \text{ and } temp = V_x\}$

$wp\ (temp = x; \{y = V_y \text{ and } temp = V_x\}) = \{y = V_y \text{ and } x = V_x\}$.

i.e. the computed precondition for the first statement is $\{y = V_y \text{ and } x = V_x\}$ which is same as the given weakest precondition. Hence the given program is correct.

3.5.3.3 EVALUATION

In the and we summarize the axiomatic semantics as :

1. Axiomatic semantics are a powerful tool for research into program correctness proofs.
2. Axiomatic semantics are based on formal and mathematical logic.

3. Axiomatic semantics cannot generally be used to define a langua completely without becoming externally complex.
4. Axiomatic semantics provide no guidance to the implementor.
5. Axiomatic semantics are appropriate for program verification a program derivation.

3.5.4 SOME OTHER SEMANTICS

In addition to operational, denotational and axiomatic semantics, there a other semantics also. These are explained below.

1. **Algebraic semantics:** The algebraic semantics describe the meaning a program by defining an algebra. The algebraic relationships an operations are described by axioms and equations.
2. **Translation semantics :** The translation semantics describe how translate a program into another language usually the language of machine. Translation semantics are used in compilers.

KEY POINTS TO REMEMBER

- The main purpose of a language is to express meaning by means of sound.
- The validity of the sentences comprising a programming language can b broken down into two things namely **syntax** and **semantics.**
- The term **syntax** refers to grammatical structure *i.e.* the syntax of programming language is the form of its expressions, statements an programming.
- The term **semantics** refers to the meaning of the vocabulary symbol arranged with that structure.
- The syntax of all programming language is defined by two set of rules **lexical rules** and **syntax rules.**
- The programming language can be formally defined in two distinct ways :
  - Language Recognition
  - Language Generation
- The various formal methods for describing syntax are :
  - BNF grammars
  - Context free grammars

  - Extended BNF
  - Syntax Graphs
  - Finite State automata
  - Regular grammar and regulator expressions
  - Push down automata.
- The process of generating the sentences of a language through a sequence of applications of the rules is called **derivation process** or **parsing.**
- Each string in the derivation is called a sentential form.
- The two basic ways by which a derivation can take place are **left most derivation (LMD)** and **Right most derivation (RMD).**
- **A content free grammar (CFG)** or BNF is ambiguous if these are two or more left most derivations of a string.
- Rules of operator precedence and associatively can be used to disambiguate the expression grammars.
- Extended BNF includes three extensions to BNF.
- The syntax graphs are used to represent the information contained in BNF and EBNF, graphically.
- A finite state automata has a starting state, one or more final states, and a set of transitions (labeled arcs) from one state to another.
- A finite state automata can be DFA or NFA.
- Regular expressions are mathematical expressions used to describe tokens.
- The type conversion is an operation which takes a data object of one type and produces the corresponding data object of a different type. The signatures of a type conversion operation is given as :

  conversion_op : type1 → type 2

- An assignment statement is one of the central constructs in imperative languages to dynamically change the binding (or association) of values to variables.
- The binding of a variable to a value at the time it is bound to storage is called initialization.
- The various numeric data types available in various programming languages are integer, real, subrange, complex numbers and rational numbers.

- An integer can be represented in storage with
  - no descriptor
  - descriptor stored in separate word
  - non-descriptor stored in same word
- The real numbers are classified as **fixed point** and **floating point** numbers.
- Subranges are a special case of basic types because they restrict the range of values of an existing type.
- A **complex number** consists of a pair of numbers, representing number's real and imaginary parts.
- A **rational number** is the quotient of two integers.
- An **enumeration** is an ordered list of distinct values.
- The range of boolean types has only two elements, one for true and one false.
- A character data type provides data objects that have a single character their value.
- The numeric ordering of the characters in the character set is called **collating sequence** for the character set.

EXERICISE

1. Define the term syntax. How it is different from semantics ?
2. Explain the general problem of describing syntax.
3. What are formal methods of describing syntax ?
4. Define the term parse tree and derivation. Also explain the concept ambiguity.
5. What is the role of operation precedence and associatively rule expression grammars ?
6. Define the term semantics. Explain the various dynamic semantics detail.
7. Write short notes on :
   (a) Language recognizer and language generation
   (b) BNF and CFG
   (c) BNF and EBNF
   (d) LMD and RMD

CHAPTER

4 STRUCTURED DATA OBJECTS

4.1 INTRODUCTION

While we have discussed elementary data types in chapter 2, in application, we find that the data is generally structured in some way. Most Imperative languages provide some support for structured types. Users may be able to define their own types, and this can create more meaningful programs. Various types can be combined to create aggregate types, composed of elements of other types, such as arrays and records. This chapter deals with specification and implementation of structured data objects.

4.2 STRUCTURED DATA OBJECTS

As described in chapter 2, a data object refers to run – time grouping of one or more piece of data in a virtual computer. If a data object is composed of other data objects, it is called **structured data object** or **data structure**. The components of structured data object can be elementary or it may be another data structure. For example, an array data object may be composed of similar elementary data objects (like integers).

The structured data objects can be

1. **System defined :** These structured data objects are set up by system itself and the programmer cannot access these data objects directly *e.g.* subprogram activation records, file buffers, run time storage stacks, Free space lists etc.
2. **Programmer defined :** These are the data objects created explicitly by programmer through declarations and statements in the programs. *e.g.* arrays, records, files etc.

4.3 STRUCTURED DATA TYPES

The data types of a language are a large part of what determines the language's style and use. Along with control structures, they form the heart of a language.

A structured data type is defined as the class of structured data objects toge with a set of operations for creating and manipulating then. The var structured data types (data structures) in a programming language can be

1. **Built-in data structures :** The Built-in data structures are built into language and are not defined in terms of other types. For example arrays. The Bu in-data structures are provided to the various programming languages because the following reasons.
   (*i*) Built-in-data structures does not need any explicit declaration by programmer. They can be directly used in the program.
   (*ii*) Built-in-data structures are directly supported by the hardware of computer.
   (*iii*) No range specification of values of data object is needed to be declared. I already defined in the language.
   (*iv*) The operations, for manipulating the data objects, are defined in language itself.
   (*v*) No declaration for the choice of storage representation has to be made the programmer. Translator itself determines the best storag representation. This reduces, overall storage requirement and executi time for the program being translated.
   (*vi*) As the range of values for the built-in-data types is already define therefore most efficient storage management procedures can be used duri execution.
   (*vii*) It eases the process of type checking and type conversion.
   (*viii*) Programmer can easily derive the various enumerated data types usi these built-in-data types.
   (*ix*) They can be directly used in writing system program for *e.g.* code computer, code for operating system etc. As built-in-data structure, a directly simulated by the hardware.
   (*x*) Built-in-data structures supported by C are arrays, structures etc.

2. **User defined data structures :** Many programming languages perm the programmer to define new data types. facilities are provided to allow t construction of any structure using pointers to link together the component da objects as desired. For example, link lists.

The two concepts regarding structured data types are
(*a*) Specification of Structured data Types, and
(*b*) Implementation of Structured Data types

4.3.1 SPECIFICATION OF STRUCTURED DATA TYPES

The basic elements of specification are
(*a*) Attributes
(*b*) Values, and
(*c*) Operations

The major attributes of structured data types are

1. **Number of Components :** Based on the number of components a data structure can be either of fixed size or of variable size.
   (*i*) **Fixed size data structure :** A data structure is called a fixed size data structure if the number of components do not change (*i.e.* invariant) during its life time *e.g.* arrays and records.
   (*ii*) **Variable size data structure :** A data structure is called a variable size data structure if the number of components vary dynamically *e.g.* stacks, lists, sets etc. The variable size data structures usually define the component insertion and deletion operations from structures.
2. **Types of each component :** Based on the type of each component comprising a data structure, a data structure can be either homogeneous or heterogeneous.
   (*i*) **Homogeneous data structure :** A data structure is homogeneous if all its constituent components are of the same type. *e.g.* arrays, character strings etc.
   (*ii*) **Heterogeneous data structure :** A data structure is heterogeneous if its constituent components are of different types. *e.g.* records and lists etc.
3. **Names to be used for selecting components :** A data structure types needs a selection mechanism for identifying individual components of the data structure. *e.g.* for an array, the name of an individual component may be an integer subscript or sequence of subscripts.
4. **Maximum number of components :** For a variable size data structure such as a character string or stack, a maximum size for the structure in terms of number of components may be specified.
5. **Organization of the Components :** The most common organization is a simple linear sequence of components. The data structures with this organization are vector, records, character strings, stacks etc. Array, record

and list types, however, also usually are extended to "multidimensiona
forms.

Operations on Data Structures : The various operations that can be carri
out on data structures are

1. **Component Selection Operations :** These are basically two types
component selection operations.

   (*a*) **Sequential component selected operations,** in which th
   components are selected in a sequential (one after the other in sequenc
   manner. *e.g.* in C if write a declaration.

   For ($i = 1$; $i <= 10$; $i++$) { ..... *a* [*i*] ..... } then we can access a
   component in a sequential manner only.

   (*b*) **Random component selection operations,** in which an arbitra
   component of the data structure can be accessed. *e.g.* in C if we write
   declaration

   int *a* [10] ;

   then we can access any component randomly *e.g.* *a* [3], *a* [7] etc.

2. **Whole data-structure operations :** Operations may take entire da
structures as arguments and produce new data structures as results. e
addition of two arrays.

3. **Insertion / deletion of components :** Operations that change th
number of components in a data structure have a major impact on storag
representations and storage management for data structures.

4. **Creation / destruction of data structures :** Operations that create an
destroy data structures also have a major impact on storage managemen
for data structures.

4.3.2 IMPLEMENTATION OF STRUCTURED DATA TYPES

The basic elements of implementation are

(*a*) Storage Representation

(*b*) Implementation of operations

(*c*) Storage management and data structures

These are described as given below.

(*a*) Storage Representation : The storage representation for a data structur
includes.

(*i*) Storage for the components of the structure.

(*ii*) An optional descriptor to store some or all of the attributes of the structure.

The two basic representations are

(I) **Sequential representation,** in which a data structure is stored in a single contiguous block of storage. This representation is shown in figure 4.1 and is mainly used for fixed size data structures and some times for homogeneous data structures *e.g.* arrays.

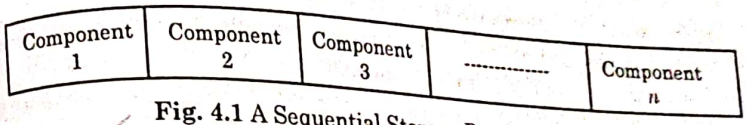


Fig. 4.1 A Sequential Storage Representation.

(II) **Linked representation,** in which a data structure is stored in several non-contiguous blocks of storage, with the blocks linked together through pointers. A pointer from one block to another block is called a link. This representation is shown in figure 4.2 and is mainly used for variable data structures such as lists.

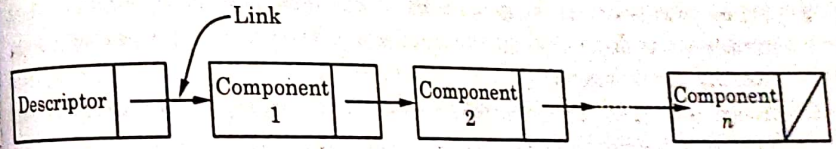


Fig. 4.2 A Linked Storage Representation

(*b*) Implementation of Operations : The implementation of operations for data structure are largely influenced by component selection operations (sequential and random). Both sequential and random selection operations are implemented differently for sequential and linked storage representation.

(*i*) **Sequential representation :** The accessing formula for random selection of a component often involves **base-address-plus-offset** calculation. The relative location of a selected component in a sequential block is called its **offset.** The starting location of the entire block is called its **base address.** Therefore to access the component A [I] involves the following accessing formula.

$$\text{Address of A [I]} = l\text{value (A)} + \text{I}$$

Here *l* value (A) represents the base address for data structure (like array) and I is the offset.

For a homogeneous structure (like array) with sequential component selection the component selection involves following steps :–

1. To select the first component using **base address-plus-offset** calculation
2. To advance to next component in the sequence, add the size of the current component to the location of the current component.

(ii) Linked representation : The random selection of a component from linked structure involves following a chain of pointers from the first block of storage in the structure to the desired component. For this selection algorithm, the position of the link pointer within each component must be known. Selection of a sequence of component proceeds by selecting the first component as above and then following the link pointer from the current component to the next component for each subsequent selection.

(c) Storage Management and Data Structure : The lifetime of a data object begins when a block of storage is allocated. The lifetime ends when the block of storage is dissolved. At the time that a data object is created, *i.e.* at the start of its lifetime, an access path to the data object must also be created so that the data object can be accessed by operations in the program. During the lifetime of a data object, additional access paths may also be created. Thus at any point during the lifetime of a data object, several access paths to it may exist. However, there are two central problems in storage management.

(i) Garbage : When all access paths to a data object are destroyed but the data object continues to exist, the data object is said to be garbage. A data object that has become garbage ties up storage that might otherwise be reallocated for another purpose.

(ii) Dangling references : A dangling reference, is an access path that continues to exist after the lifetime of the associated data object. The dangling references may compromise the integrity of the entire run-time structure during program execution. It may also modify storage that has already been allocated to another data object of entirely different type or it can modify housekeeping data (such as a link to a free space list) that has been stored there temporarily by the storage management system.

Due to the problems of garbage and dangling references as discussed above, in most languages, operations that create and destroy data structures and that insert and delete components of data structure are designed with a careful eye toward their effect on storage management. Often the result is a tight set of restrictions on such operations.

5.4 DECLARATIONS AND TYPE CHECKING FOR DATA STRUCTURES

A declaration in a program refers to a statement that provides the information regarding, the name and type of structured data object (data structure) to the programming language translator. As in case of data structures there are more attributes to be specified, therefore structures are generally more complex than elementary data objects. The declaration of a data structure *e.g.* array provides the following information :

1. Data type is an array
2. Number of dimensions
3. Subscripts naming the rows and columns
4. Number of components
5. Data type of each component

For example if we declare an array in Pascal as

A : array [1 ... 10, – 5 ... 5] of real;

then
(1) Data type is array
(2) It is two dimensional array
(3) Subscripts naming the rows are integers from 1 to 10 and the subscripts naming the columns are integers from – 5 to 5.
(4) Total no. of components are 110.
(5) data type of each component is real.

The purposes for declarations have already been explained in chapter 2 section 2.5.1. Note that, without the declaration, the attributes of A would have to be determined dynamically at run time, with the result that the storage representation and component accessing would be much less efficient.

Type Checking : The basic concept of type checking for data structure are similar to those discussed for elementary data objects however type checking is somewhat complex for data structures because component selection operations must be taken into account. Two main problems in type checking for data structures are.

1. **Existence of a selected component :** The arguments to a selection operation may be of right type, but the designated component may not exist in the data structure *e.g.* component selection operation in an array may receive a subscript that is out of bound and produces invalid *l*-value. The effect is similar to type-checking error. Therefore run-time checking may be

required to determine whether the selected component exists before
formula is used to determine its precise *l*-value.

2. **Type of a selected component :** A selection sequence may defin
complex path through a data structure to the desired component. Howev
it cannot in general be assumed that the selected component exists wh
needed at run time.

4.5 VECTORS AND ARRAYS

An array is a homogeneous aggregate of data elements in which an individ
element is identified by its position in the aggregate, relative to the first element
reference to an array element in a program often includes one or more non-consta
subscripts. Such references require a run-time calculation to determine the memo
location being referenced.

Specific elements of an array are referenced by means of a two-level syntac
mechanism, where the first part is the aggregate name, and the second part i
possibly dynamic selector consisting of one or more items known as subscripts
indexes. An array type specifies the index of the first and last elements of the arr
and the type of all the elements. The index of the first elements is called the low
bound and the index of the last element is called the **upper bound** of the array
all the indexes in a reference are constants, the selector is static ; otherwise it
dynamic. Based on the number of dimensions an array can be :

1. One-dimensional array or Vector
2. Two-Dimensional array
3. Multi-Dimensional array

These are described as given below :

1. **One-Dimensional Array or Vector :** A vector is a data structu
composed of a fixed number of components of the same type organized as
simple linear sequence. A component of a vector is selected by giving
subscript, indicating the position of the components in the sequence.

Specification : The specification of vectors is as discussed below –

(a) Attributes : The main attributes of a vector are

(*i*) Number of components

(*ii*) Data type of each component

(*iii*) Subscript to be used to select each component. It includes provid
lower and upper bounds.

For example consider an array declaration in C.

int *a* [10] ;

The above declaration specifies then *a* is an array (or vector) of 10 integer components with subscripts ranging from 0 to 9.

The array declaration in Pascal provides both lower as well as upper bound explicitly *e.g.*

a : **array** [– 5 ..... 5] **of integer** ;

(b) Values : The values for vectors would be the set of numbers that form valid values for array components. Imperative languages favour values that can be used in machine locations, so whole array values appears in specialized contexts. An example is array initialization in C. An array initializer is a sequence of values for array elements *e.g.*

int coin [ ] = {1, 5, 10, 25, 50, 100}

(c) Operations : An array operation is one that operates or an array as a unit. Some languages, such as FORTRAN 77, provide no array operations. However, a variety of programming languages provides following operation on vectors.

(*i*) **Subscripting Operations :** Specific elements of an array are referenced by means of a two-level syntactic mechanism *i.e.* aggregate name followed by a possibly dynamic selector consisting of one or more items known as subscripts or indexes. If all of the indexes in a reference are constants, the selector is static ; otherwise it is dynamic. The selection operations can be thought of as a mapping from the array name and the set of index values to an element in the aggregate. Indeed, arrays are sometimes called finite mappings. Symbolically, this mapping can be shown as

array_name (index_value_list) → element

e.g. *a* [7]. However, the subscript may generally be a computed value, in which case an expression may be given that computer the subscript *e.g.*, *a* [I + 2].

(*ii*) To create and destroy vectors.

(*iii*) Assignment to components of a vector *e.g.* Ada allows array assignments, including those where the right side is an aggregate value rather than an array name.

(iv) Arithmetic operations on the pairs of vectors of the same size, addition of two vectors in which the corresponding eleme (components) are added.

(v) As vectors are of fixed size, insertion and deletion of components not allowed.

(vi) FORTRAN 90 includes a number of array operations that are cal elemental because they are operations between pair of array eleme For example, the add operator (+) between two vectors results in vector of the sums of the element pairs of the two arrays.

(vii) Some programming languages like APL also includes several spec operators that take other operators as operands. One of these is inner product operator, which is specified with a period (.). It ta two operands, which are binary operators. For example, +.X is a n operator that takes two vectors. It first multiplies the correspond elements of two arguments, and then in sums the results.

Implementation : Implementing arrays requires more compile ti effort than does implementing simple types, such as integer. The co to allow accessing of array elements must be generated at comp time. At run time, this code must be executed to produce eleme addresses. There is no way to precompute the address to be access by a reference such as A [I].

A single dimensioned array (a vector) is a list of adjacent mem cells. Beginning at the initial component, the I^{th} component can addressed by skipping (I – 1) components. If E is the size of ea component, then we must skip (I – 1) × E memory location. (assumi a lower bound CB = 1). Therefore

$$\boxed{l \text{ value (A [I])} = \alpha + (I-1) \times E}$$

where α = The location at which the first element of the vector beg and LB = 1.

If we generalize the value of lower bound to LB then the abo formula can be written as

$$l \text{ value (A (I)]} = \alpha + (I - LB) \times E = (\alpha - LB \times E) + I \times E = K + I \times E$$

where $K = \alpha - LB \times E$ and is a constant.

If we consider the example of character arrays in C then

$$E = 1, \quad LB = 0$$

Therefore $\boxed{l \text{ value (A[I])} = \alpha + I}$

Let us now address the element with subscript 0 of our vector.

$$l\text{value (A [0])} = (\alpha - LB \times E) + (0 \times E) = \alpha - LB \times E = K$$

i.e. K represents the address that the element 0 of the vector would occupy, if it existed. Since the zeroth element may not be part of the array (since the array may have lower bound greater than 0), this address ia called the **virtual origin (VO).**

The concept of virtual origin (VO) gives an algorithm for building vectors and generating the access formula.

1. Allocate storage for a vector consisting of N components each of size E and a descriptor of size D *i.e.* allocate D + N × E memory locations.
2. Compute the virtual origin VO = α – LB × E
3. Calculate the *l*-value of A [I] as

$$\boxed{l\text{value (A(I)]} = VO + I \times E}$$

Note that if the subscript I is not within the range LB ≤ I ≤ UB, storage can never be accessed as an *l*-value, which does not represent a valid subscript.

If the virtual origin is stored in the array descriptors, then the actual array need not be contiguous with its descriptor as shown in Figure 4.3.

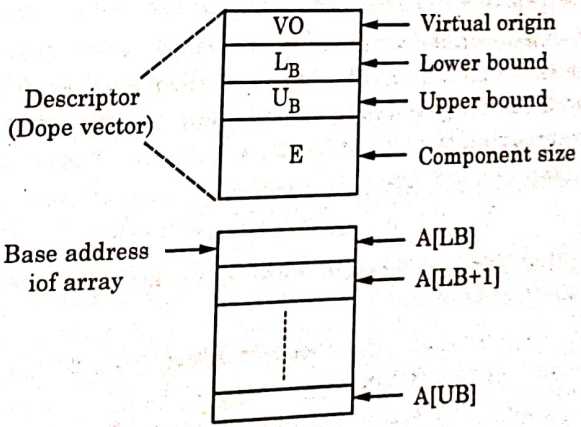


Fig. 4.3 Separate Storage Area for Descriptor and Components.

This is the usual case where descriptors for array parameters may be passed to subprograms, and the actual array storage stored elsewhere.

Packed and Unpacked Storage Representations : A packed storage representation is one in which the components of a vector are packed into storage

sequentially without regard for placing each component at the beginning of addressable word or byte of storage. A packed storage representations results substantial savings in the amount of storage required for a vector. However, access to components of a packed structure is expensive. More complex series calculations are required and if a component may cross a word boundary, then acc is even more difficult.

Due to the disadvantages of packed representations, vectors are often stored unpacked form *i.e.* each component is stored beginning at the boundary of addressable unit of storage and between each pair of components there is unu storage that represents **padding.** The main advantage here is the easier acc using a simple formula but at a cost in lost storage. However, with the growth byte-addressable machines and the subsequent decline of word addressa machines, this is rarely an issue in most systems today.

Whole Vector Operations : The whole vector operations are the operati that work on entire vectors as a unit *e.g.* assignment of one vector to anot arithmetic operations etc.

A major implementation problem with such whole – vector operations conce the storage required for the result. Storage must be allocated temporarily to st the *r*-value of this result, unless it is immediately assigned to an existing *l*-va location. This may also increase the complexity and the cost of program execution.

2. Two-Dimensional arrays or matrices : A two-dimensional g composed of rows and columns of components is called a 2-D array or a matrix.

Specification : The attributes, values and operations for vectors can generalized to 2-D arrays. The only difference is that a subscript range for both dimensions is required. *e.g.* consider the following C declaration.

int *a* [5] [4] ;

It creates an integer variable, *a*, which is an array of five elements, each hich is an array of four elements.

$$\begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \\ a_{40} & a_{41} & a_{42} & a_{43} \end{bmatrix}$$

Implementation : A 2-D array can be conveniently considered as a vector vectors. The 2-D arrays are more complex to implement than 1-D array (or vect The hardware memory is linear – it is usually a simple sequence of bytes. So,

values of data types that have two or more dimensions must be mapped onto the single dimensioned memory. There are two common ways in which the 2-D arrays can be mapped to one dimension.

(*a*) Row major order

(*b*) Column major order

(*a*) Row major order : In a row major order, the matrix is considered as a vector in which each element is a sub vector representing one row of the original matrix. In row major order, the elements of the array that have as their first subscript the lower bound value of that subscript are stored first, followed by the elements of the second value of the first subscript, and so fourth. This is also known as **column-of-rows** structure. *i.e.* if the array is matrix, it is stored by rows. For example, if the matrix had the values.

1 4 5
2 9 3
7 8 6

It would be stored in row major order as :

1, 4, 5, 2, 9, 3, 7, 8, 6.

(*b*) Column-Major –order : In a column-major-order representation, the matrix is treated as a single row of columns. In column major order, the elements of an array that have as their last subscript the lower bound value of that subscript are stored first, followed by the elements of the second value of the last subscript, and so forth. If the array is a matrix, it is stored by columns. If the example matrix above were stored in a column major order, it would have the following order in memory.

1, 2, 7, 4, 9, 8, 5, 3, 6

The column major order is used in FORTRAN, but the other languages use row-major order.

Storage representation : The storage representation for a 2D array (matrix) directly follows from that for a vector. The data objects are stored assuming a row major order in a single sequential block of memory containing all the components of the array in sequence. The descriptor for a 2D array is same as for 1 –D array except that an upper and a lower bound for the subscript range of each dimension are needed. The storage representation for a sample 2-D array is as shown in Figure 4.4.

Access Formula to compute the offset of a Component

Consider a 2-D array as shown below.

A : array [LB_1 ..... UB_1 , UB_2 ..... UB_2] of data type ;

Here LB_1 and LB_2 are lower bounds on subscript 1 and subscript respectively and UB_1 and UB_2 are upper bounds on subscript 1 and subscript respectively.

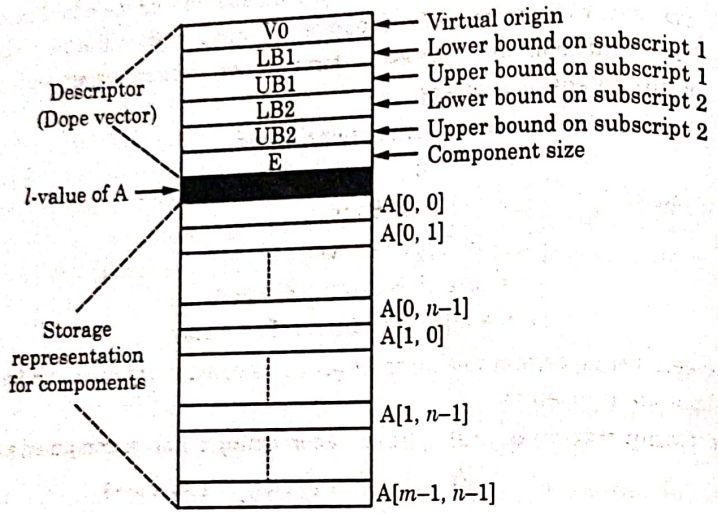


Fig. 4.4. Storage Representation for 2-D Array.

To find the *l*-value of A [I, J], first determine the number of rows to skip over ($I - LB_1$), multiply by the length of a row to get the location of the start of I^{th} row and then find the location of the J^{th} component in that row, as for a vector, Thus, A is a matrix with M rows and N columns. There are two possible accessing formulas.

If the elements of A are in row major order then

$$l\text{-value}(A[I, J]) = \alpha + (I - LB_1) \times S + (J - LB_2) \times E$$

where α = base address

S = length of row = $(UB_2 - LB_2 + 1) \times E$

LB_1, LB_2 = Lower Bounds on first and second subscripts respectively

UB_1, UB_2 = Upper Bounds on first and Second subscripts respectively

Therefore, $S = (UB_2 - LB_2 + 1) \times E$

$$VO = \alpha - LB_1 \times S - LB_2 \times E$$

$$\boxed{l\text{value}(A[I, J]) = VO + J \times S + J \times E}$$

Similar to the vector case, E = component size

S = Size of each row of the matrix

VO = Virtual origin

3. **Multi-Dimensional arrays :** The 1-D arrays (vectors) and 2-D arrays (matrices) can be generalized to any number of dimensions. For example, a 3D array is composed of planes of rows and columns. The storage representation and accessing formula for vectors generalize readily to multidimensional arrays.

Assume we have an n – dimensional array

$$A[L_1 : U_1, \ldots\ldots L_n : U_n]$$

where $L_i = i^{th}$ Lower Bound

$U_i = i^{th}$ Upper Bound

e = Size of each array element

α = Beginning address of the array

The general algorithm to allocate descriptors in array creation and for accessing array elements is given as

(a) Computation of multipliers : Each multiplier m_i is computed as follows :

$$m_i = \begin{cases} (U_{i+1} - L_{i+1} + 1) \times m_{i+1}, & 1 \le i \le n-1 \\ e, & i = n \end{cases}$$

(b) Computation of Virtual Origin (VO)

$$VO = \alpha - \sum_{i=1}^{n} (L_i \times m_i)$$

The multipliers m_i and the virtual origin VO can be stored in the run time descriptors for the array.

(c) Address of array element : The address of element A [S_1 , S_2 ..... S_n] is given by

$$\boxed{l\text{value}\left(A[S_1, S_2, \cdots S_n]\right) = VO + \sum_{i=1}^{n} (s_i \times m_i)}$$

4.5.1 SLICES

Specification : A slice is a substructure of an array that it is itself an array. For example, if A is a matrix, the first row of A is one possible slice,. as are the last row and the first column. It is important to realize that a slice is not a new data type. Rather, it is a mechanism for referencing part of an array as a unit. If arrays

cannot be manipulated as units in a language, that language has no use for slices. The examples of slices are shown in Figure 4.5.

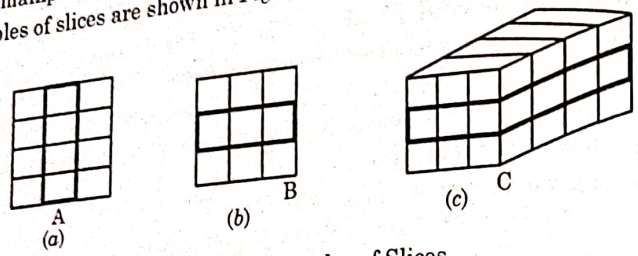


Fig. 4.5 : Examples of Slices.

(*a*) : A column slice representing column two if a three-column matrix.

(*b*) : A row slice representing a row two of a 3-row matrix.

(*c*) : A multidimensional slice representing second plane of a 3-D array.

PL/I was one of the earliest languages to implement slices. The other languages that implement the concept of slices are FORTRAN 77, FORTRAN 90, Ada etc.

Implementation : The use of descriptors permits an efficient implementation of a slice *e.g,* the 3-by-4 array A can be described by the descriptor :

VO	α
LB1	1
UB1	4
Multiplier 1	3
LB2	1
UB2	3
Multiplier 2	1

Fig. 4.6. The Descriptor For a Slice.

Accessing Formula to compute the offset of a component

The address of A [I, J] *i.e.*

l-value (A [I, J]) = $\alpha + I \times 3 + J \times 1$

The multiplier 2, which also represents the size of the data object, is also the distance between successive elements of the array. In this case, the elements are contiguous. However, this not need be true. A slice has the property that the elements of 1-D may not be contiguous, but are equally spaced. Therefore, we can

represent the slice A (*, 2) as a descriptor, based upon the descriptor for A as follows :

VO	$\alpha - 2$
LB1	1
UB1	4
Multiplier	3

Fig. 4.7. Descriptor for slice of A.

The figure 4.7 represents a 1-D vector of length 4, starting one location after A, with each element three locations apart. We can represent slices for B and C in an analogous manner.

4.6 RECORDS

There is frequently a need in programs to model collections of data that are not homogeneous. For example, information about a college student might include name, roll number, age, marks and so forth. A data type for such a collection might use a character array for the name, an integer for the roll number, age and marks and so forth. Records are designated to meet this kind of need.

Specification : The record structure was first introduced in COBOL and has been common in programming languages since then. It permits the grouping of information kept on a particular item.

A **record** is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names. Records allow variables relevant to an object to be grouped together and treated as a unit.

Records and arrays : Both records and arrays are forms of fixed-length linear data structures, but records differ from array in the following ways.

1. The components of arrays are homogeneous while the components of records may be heterogeneous.
2. In arrays the components are selected using expressions that may be evaluated at run time while in records the components are named with symbolic names known as compile time. The above comparison is summarized in table 4.1 as given below.

	Arrays	Records
Component Types	Homogeneous	Heterogeneous
Component selectors	Expressions evaluated at run time	Names known at compile time

Table 4.1 : Comparison between arrays and records

Attributes : The main attributes of a record are

1. The number of components
2. The data type of each component
3. The selector used to name each component

For example consider the following C declaration.

```
struct student
 {
    int    roll ;
    char   name [10] ;
    int    age ;
 } s ;
```

The above declaration defines a record of type student consisting of following attributes.

1. The number of components are 3.
2. The data types of components are integer, character array and integer respectively.
3. The components are selected using a variables declared to be of type student as

```
s . roll
s. name [i]
s. age
```

Note : The components of a record are also called fields, and the component names are also called field names. Records are also called structures in some programming languages like C, C++ etc. Java does not have struct.

Operations : The following operations can be applied on structures.

1. **Assignment :** The assignment is a common record operation. In most cases, the types of the two sides must be identical *e.g* For above structure declaration consider

```
struct student s1, s2 ;
.....
s2 = s1 ;
```

Here the s2 has the same attributes as s1. The correspondence of component names between records is also made the basis for assignment in COBOL and PL/I *e.g.* in the COBOL.

MOVE CORRESPONDING s1 to s2

which assigns each component of s1 to the corresponding component of s2, where corresponding components must have the same name and data type but need not appear in the same order in each record.

2. **Component Selection :** The components of records are named with symbolic names. The components are selected using a variable declared of structure type. *e.g.*
3. The two records can be compared for equally in an if statement.

Implementation : The fields of records are stored in adjacent memory locations. Since the storage for each field is known, the offset to each component can easily be computed. The storage allocation of a record made as a contiguous block for each field is shown in figure 4.8.

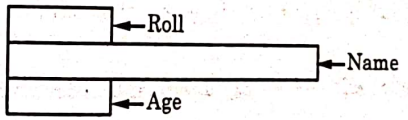


Fig. 4.8 Storage Representation for Struct Student.

Accessing Formula for component selection

In records the sizes of fields are not necessarily the same, the access method used for arrays is not used for records. Instead, the offset address, relative to the beginning of the records, is associated with each field. Field accesses are all handled using these offsets. The compile time descriptors for a record has the general form shown in Figure 4.9.

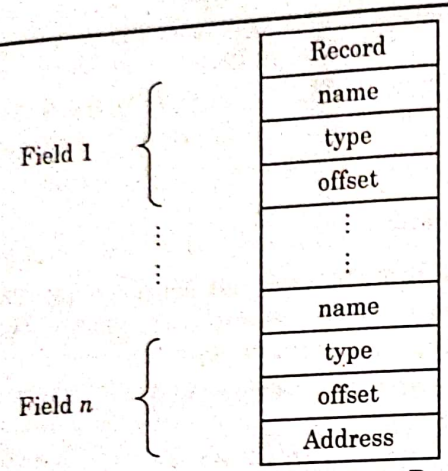


Fig. 4.9. A Compile Time Descriptors for a Record.

The basic accessing formula used to compute the location of the I^{th} component is :

$$l \text{ value } (R.I) = \alpha + \sum_{J=1}^{I-1} (\text{size of } R.J.)$$

where α = Base address of the storage block representing R.

R.J. = J^{th} component

However, the summation may always be computed during translation to give the offset, K_I, for the I^{th} component, so that during execution only the base address for the storage block need be added :

$$l \text{ value } (R.I) = \alpha + K_I$$

Array of records : Some programming languages like C allows the arrays in which each component is a record. *e.g.*

```
struct student
{
    int roll ;
    char name [10] ;
    int age ;
} s [100] ;
```

The above declaration declares a 1-D array of 100 components, each of which is an student record. A component of such a composite data structure is selected by

using a sequence of selection operations to select first a component of the vector and then a component of the record as in s[16].age.

A record may also have components that are arrays or other records, leading to records that have hierarchical structure consisting of a top level of components, some of which may be arrays or records. For example the second component name is an array of characters.

Implementation : The storage representations developed for simple vectors and records extend without change to vectors and records whose components are themselves vectors or records. An array of records has the same storage representation as a vector of integers or any other elementary type, except that the storage block representing a component in the large block representing the vector is the storage block for a record. Similarly, a record whose components are records or vectors retains the same sequential storage representation, but with each component represented by a sub-block that may be itself the representation of an entire record. Selection of the components requires only a sequence of selections starting from the base address of the complete structure and computing an offset to find the location of the first-level component, followed by computation of an offset from this base address to find the second-level component, etc.

4.6.1. VARIANT RECORDS

Specification : The records are used for representing objects with common properties; all record of the same type have the same fields in common. Variant records are used for representing objects that have some but not all properties in common. The variant records have a part common to all records of that type, and a variant part, specific to some subset of the records.

Starting with Pascal, it became common to use variant records. The variant part may occur at the end of record declaration consider the following Pascal declaration.

```
type
    employee Rec = record
            name : string [25];
case salaried : boolean of
            true :(salary : real;
                union member : boolean);
            false : (hourlyRate : real;
                hours worked : red)
            end; {record}
```

var employee : employeeRec;

The tag field **salaried** permits discrimination of the type of data kept for salaried employees from those of hourly employees.

Component selection operation : The component selection operation for components of a variant record is same is that for an ordinary record, e.g. employeeRec.salary. For ordinary records, each component exist throughout the lifetime of the record but in variant record, the component may exist at one point during execution and may later disappear during another point of execution. The problem of non-existence component selection of variant record has the following possible solutions.

(i) Dynamic checking

(ii) No checking

Implementation : The storage allocated for a variant record must be sufficient for the largest of the records to be stored, and record descriptors for each of the variants must be maintained. The storage allocation for a variant record discussed earlier is shown in figure 4.10.

name	
Salaried	
Salary	Hourly rate
Union member	Hours worked

Fig. 4.10 : Storage Allocation for Variant Record.

Here the storage required for the Boolean Field Union Member is less than the needed for the real field hours worked in this case, while salary and hourly rate are both real.

When no checking is done then for selecting a component of a variant record, the offset of the selected component with in the storage block is computed during translation and during execution, the offset is added to the base address of the block to determine the location of the component. If that location is currently being used as part of a component in the current variant, then unpredictable changes are made to the value of that component.

If dynamic checking is done for component selection, then at run time the base address-plus-offset calculation is same but first the value of base field is checked to ensure that the proper variant currently exists.

4.7 CHARACTER STRINGS

Specification : A string is defined as a finite sequence of alphanumeric characters. A special character is placed at the end of every character string. This special character, null is written as "\0". It is automatically appended at the end of the characters in a string constant when they are stored. This eliminates the need to specify the number of characters that are contained inside a characters string. Apart from the null character, there are other escape sequences that can be used in a string.

Character string constants are used to label output, and input and output of all kinds of data is often done in terms of strings. Of course, character strings also are an essential type for all programs that do character manipulation.

A number of languages, including Java, incorporate strings as a primitive type, and this is probably most convenient. For the use, In Pascal, Ada, C and C++, however, the character is the primitive type, so, strings must be stored as array of characters. In Pascal, they must be stored as packed arrays in order to permit lexical comparisons. An example of using a string in C involves following declaration.

char *a* [16] ;

Several approaches can be taken to maintaining the length of a string. Three will be considered here.

1. **Static (Fixed) string length :** A character string data object may have a fixed length that is declared in the program. This can be implemented as a contiguous block of storage. For the number of characters specified. Strings must match exactly the size declared, so truncation or padding may be needed. If the shorter strings are desired, the array may be partially filled, but the programmer must keep track of the number of characters used. This technique is used in languages like Pascal, COBOL and Ada. *e.g.* in Pascal.

   name : **packed array** [1 – 10] **of char** ;

   declares a character string variable name containing a string of 10 characters.

2. **Variable length scheme with a fixed maximum :** The language PL/I uses this technique. In this technique a fixed maximum string length is declared in the program, but the actual value stored may be a string of shorter length. In this case, longer strings are truncated, and the compiler keeps track of the number of characters filled.

3. **Dynamic (Unbounded) string length :** The language SNOBOL4 allo... this technique. In this technique, a character – string data object may ha... a string value of any length, and the length may vary dynamically duri... execution with no bound (with in certain memory limits). This is certain... more convenient to use, but more system overhead is required. Either ... linked list of characters is needed, or strings would have to be stored ... dynamic memory in the heap.

String Operations : A variety of string – handling operations is often usef... Some of these operations are discussed below.

1. **Concatenation :** String concatenation forms a longer string by joining t... strings. For example, 'program' + 'ming' forms the string 'programming'. Also, if || is the symbol used for the concatenation then 'program' || 'ming' gives 'programming'. When a language supports only static strin... lengths, some caution must be taken if the result is to be stored in a stri... variable.

2. **Pattern matching :** Pattern matching is a fundamental string operation. It is often provided by a library function rather then as an operation in th... language. SNOBOL4 has an elaborate pattern matching operation bui... into the language. SNOBOL4 is probably the ultimate string manipulati... language. For example

pos ('I', 'hello')

returns 3, the first position of the letter 'I' in the string 'hello'. Som... languages like Java, do not include string-handling functions directly, b... provide a package that includes a string class and methods f... manipulating them.

3. **Substring selection :** Many languages provide an operation for selecti... a substring by giving the positions of its first and last characters. F... example,

substr (name, 1, 10)

extracts the first 10 characters of name. Another approach is to use slices ... which,

name [1 .. 10]

performs the same function. Slices, are supported in Ada.

4. **Relational Operations :** The relational operations like equal, less than, greater than etc. may be extended to strings. This requires the use of lexicographic ordering between strings in which a string A comes before string B in lexicographic order if

(*a*) Either the first character A is less than the first character of B, or

(*b*) The first characters are equal, and the second character of A is less than the second character of B and so on.

5. **Input-Output formatting :** The string operations are provided for breaking up formatting input data into smaller data items or for formatting output data. For example, in FORTRAN and C, the extensive sets of operations are provided for this purpose.

Implementation : As mentioned earlier, there are three different methods for handling characters strings. each method utilize a different storage representation.

1. **Static (Fixed) String length :** A descriptor for a static characters string type, which is only required during compilation has three fields as shown in Figure 4.11 (*a*).

Static string
Length
Address

Fig. 4.11 (*a*) Compile Time Descriptor for Static Strings.

The first field of every descriptors is the name of the type. The second field is the type is length (in characters). The third field is the address of the first character. The storage representation for a sample fixed length string is shown in Figure 4.11 (*b*). Strings stored 4 characters/word padded with blanks.

P	R	O	G
R	A	M	M
I	N	G	

Fig. 4.11 (*b*) Storage Representation for Fixed Length String

2. **Variable length scheme with a fixed maximum :** This technique for handling character strings require a run-time descriptors to store both the fixed maximum length and the current length as shown in figure 4.12 (*a*). The storage representation for a sample string is shown in figure 4.12 (*b*).

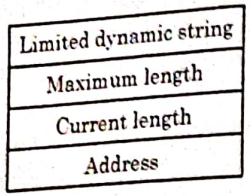


Fig. 4.12 (*a*) Run-time Descriptor For Limited Dynamic Strings.

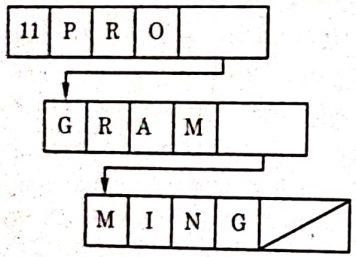
Current and maximum length stored at the header of string

11	14	P	R
O	G	R	A
M	M	I	N
G			

Fig. 4.12 (*b*) Storage Representation For Variable Length String With Bound

3. **Dynamic (Unbounded) String length :** The dynamic length strings require more complex storage management. The length of a string, and therefore the storage to which it is bound, must grow and shrink dynamically. These are two possible approaches to the dynamic allocation problem.

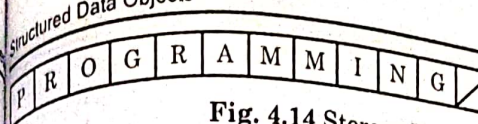
(*a*) **Unbounded string length with fixed allocations :** In this approach, the strings can be stored in a linked list, so that when a string grows, the newly required cells can come from anywhere in the heap. The storage representation for a sample string is shown in figure 4.13. String stored at 4 characters/block length at header of string.



String stored at 4 characters/block. Length at header of string

Fig. 4.13 Storage Representation For Unbounded String Length With Fixed Allocations.

(*b*) **Unbounded string length with variable allocations :** In this approach, the complete string is stored in adjacent storage cells. This is the basic technique used in C. The storage representation for a sample string is shown in figure 4.14.



String stored as contiguous array of characters, terminated by null character.

Fig. 4.14 Storage Representation For Unbounded String Length With Variable Allocations.

The problem with this method occurs when a string grows : How can storage that is adjacent to the existing cells continue to be allocated for the string variable ? Frequently, such storage is not available. Instead, a new area of memory is found that can store the complete new string and the old part is moved to this area. Then the memory cells used for the old strings are deallocated.

Implementation of operations : Character string types (fixed length representations) are sometimes supported directly in hardware, but in most cases software is used to implement string storage, retrieval, and manipulation operations. When character, string types are represented as character arrays, the language often supplies few operations. Operations on strings such as concatenation, substring selection and pattern matching are ordinarily entirely software simulated.

4.8 VARIABLE SIZE DATA STRUCTURES

A data structure is called a **variable size data structure** if the number of components changes dynamically. Variable size data structure types usually define operations that insert and delete components from structures. The examples of variable size data structure, include lists, stacks, sets, table and files etc. Variable – size data objects often use a pointer data type that allows fixed – size data objects to be linked together explicitly by the programmer. Here, in this section we discuss the lists in detail while the other variable size data structure are discussed in the following sections.

4.8.1 LISTS

A data structure composed of an ordered sequence of data, structures is usually termed as a **list**.

Specification : Lists are similar to arrays (vectors) in that they consist of an ordered sequence of objects. The first member of the list is called the **head** and the other members are called **tail** of the list. The lists differ from vectors in following respects :

1. Vectors are of fixed length while lists are rarely of fixed length.
2. Lists can grow and shrink during program execution.
3. Vectors are homogeneous but lists are rarely homogeneous.

4. Languages using lists declare such data implicitly without explicit attributes for list members.

The declarative languages like LISP and PROLOG include a list type. The entries in lists can be either items (called atoms) or other, lists. For example consider the following LISP list structure.

(Function Name Data 1, Data 2, ... Data *n*)

LISP executes by applying function name to arguments Data 1 through Datan.

Operations : Operations in lists include the ability to construct and disassemble lists. Most operations take list arguments and return list values. The following primitives are provided in LISP for component selection.

1. **CAR (content of the address register) :** Given a list L as operand, the (CAR L) return a pointer to the first list element (head) *e.g.* (car (*a b*)) = *a*
2. **CDR (Content of the decrement register) :** Given a list L as operand, the (CDR L) returns a pointer to the list with the first element deleted (tail of the list). *e.g.* (CDR (*a b*)) = (*b*). As another example if L = (A B C) then (CAR (CDR L)) = (B)
3. **The CONS primitive :** The cons primitive takes two pointers as operands, allocates a new list element memory word, stores the two pointers in the car and cdr fields of the word, and return a pointer to the new word. When the second operand is a list the effect is to add the first element to the head of this list and return a pointer to the extended list. For example (CONS a (b cd) = (ab cd). Also (CONS (a b c) (d e f)) = ((a b c) d e f)

Another operations include, quote, replace a, nuts etc.

Implementation : The two features common to declarative languages like LISP are

(*a*) Each data object carries a run-time descriptor giving type and other attributes.

(*b*) If a data object has components (*i.e.* is a structured data object), then the components are almost never represented directly as part of the data object; instead a pointer to the component data object is used.

Lists are usually stored as a single linked list structures. The link list represent ts in LISP has three fields as shown in figure 4.15.

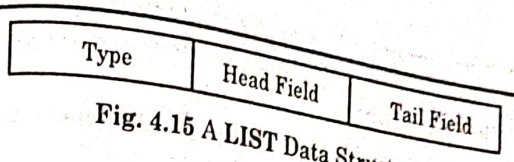


Fig. 4.15 A LIST Data Structure.

Each node has two pointers and represents an element A node for an atom has its first pointer pointing to some representation of the atom, such as its symbol or numeric value. A node for a sublist element has its first pointer pointing to the first node of the sublist. In both cases, the second pointer of a node points to the next element of the list. A list is referenced by a pointer to the its first element. The internal representation for two example lists are shown in figure 4.16.

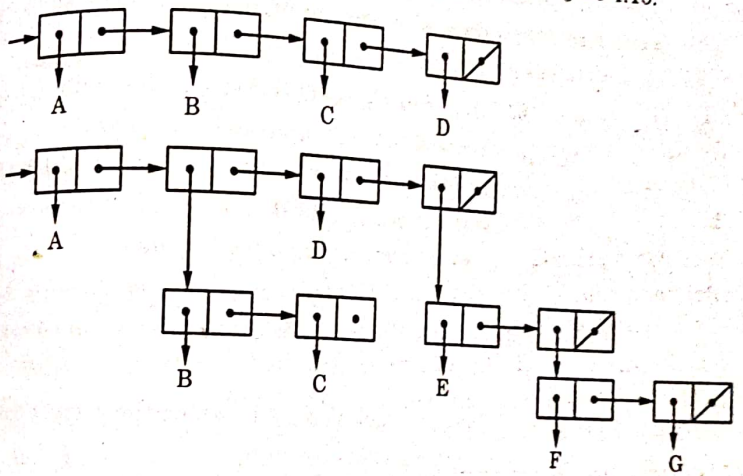


Fig. 4.16 Internal Representation of two LISP Lists
(*a*) (A B C D) (*b*) (A (B C) D (E (F G)))

Note that the elements of a list are shown horizontally. The last element of a list has no successor, so its link is NIL. Sublists are shown with the some structure.

Implementation of Operations : LISP was initially implemented in the early 1960s on an IBM 704 computer. This was a 36-bit word machine where upper is bit were called **address register** and lower 18 bits were called **decrement register.** Therefore, a head operation is implemented as CAR operation while tail operation as CDR. The operation CONS, CAR and CDR are implemented as.

1. **CONS :** A CONS or a join operation is implemented by operating a new list node and making the head field the first argument of the coins and making the tail field the second argument of the CONS.

2. **CAR :** The head of a list is the contents (r-value) of the head field of the l... item.
3. **CDR :** The tail of a list is the contents (r-value) of the tail field of the l... item.

4.8.2 VARIATIONS ON LISTS

Several variations on lists are

1. **Stacks :** A stack is a list in which **LIFO (Last In First Out)** concept ... used *i.e.* the item inserted last is the first one to be removed. In stack, ... the operations like insertion, deletion and retrieval are restricted to one e... called the TOP of the stack. The storage representation for stack can ... sequential or linked.
2. **Queues :** A queue is a list in which **FIFO (First In First Out)** concept ... used *i.e.* the item inserted first is the first one to be removed. In queue, t... component insertion takes place at one end of the queue called **REAR** a... the deletion takes place at the other end called **FRONT** of the queue. T... storage representation for queue can be sequential or linked.
3. **Trees :** A tree is a list in which components may be lists as well ... elementary data objects provided that each list is only a component of ... most one other list.
4. **Directed graphs :** A directed graph is a list in which the components m... be linked together using arbitrary linkage patterns.
5. **Property Lists :** A **property list** is defined as the record with vary... number of components if the components may vary without restrictions. ... property list, both the component names and their values must be store... each component name is called property name and the corresponding val... is called property value. A common representation is linked list as shown ... figure 4.17. A particular property value can be selected by searching ... property list until the desired property is found. The next list compon... gives the corresponding property value. The concept of property list ... provided directly in the languages like ML and LISP.

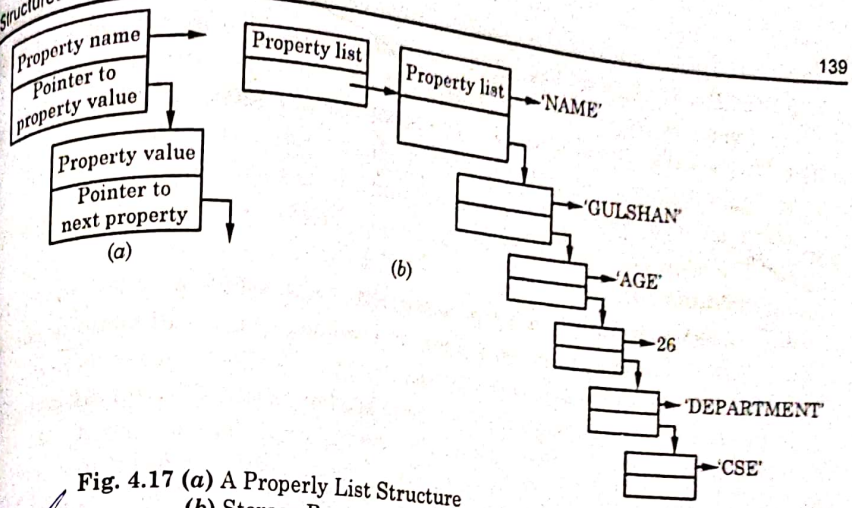


Fig. 4.17 (*a*) A Properly List Structure
(*b*) Storage Representation of a Sample Property List.

4.9 UNIONS

Specification : If it is desired or necessary to store more than one type of value at the same location, it may be possible to use a union type. A **union** is a type that may store different type values at different times during the program execution. As an example of the need for a union type, consider a table of constants for a compiler, which is used to store the constants found in a program being compiled. One field of each table entry is for the value of the constant. Suppose that for a particular language being compiled, the types of constants were integer, floating point, and Boolean. In terms of table management, it would be convenient if the same location, a table field, could store a value of any of these three types. Then all constant values could be addressed in the same way. The type of such a location is, in a sense, the union of the three value types it can store.

Type checking of unions requires that each union construct include a type indicator, called a **tag** or **discriminant,** and a union with a discriminant is called a **discriminated union.** In language that permit tags to be omitted, they are known as **free unions.** The languages like FORTRAN, C and C ++ provide union constructs called free unions. Consider the following declaration in

```
union
 {
   char name [25] ;
   int  idno ;
   float salary ;
 } emp ;
```

Operations : A union has all the operations supported by a structure except for minor changes which are a consequence of the memory sharing properties of union. This is made evident by the following operations on unions which are legal.

1. An union variable can be assigned to another union variable.
2. A union variable can be passed to a function as a parameter.
3. The address of the union variable can be extracted by using the address operator (&)
4. A function can accept and return a union or a pointer to a union.
5. Performing operations on unions as a whole, *e.g.* arithmetic or logic operations are illegal, as in structures.

Implementation : Discriminated unions are implemented by simply using the same address for every possible variant. Sufficient storage for the largest variant is allocated. The tag of the discriminated union is stored with the variant in a record like structure.

At compile time, the complete description of each variant must be stored. This can be done by associating a case table with the tag entry in the descriptor. The case table has an entry for each variant, which points to a descriptor for that particular variant. To illustrate this arrangement, consider the following Ada example :

```
type NODE (TAG : BOOLEAN) is
  record
    case TAG is
      when true ⇒ COUNT : INTEGER ;
      when false ⇒ SUM : FLOAT ;
    end case ;
    end record ;
```

The descriptor for this type could have the form shown in figure 4.18.

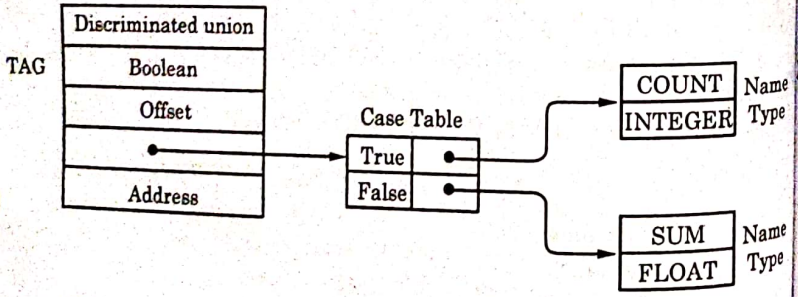


Fig. 4.18 A Compile-Time Description for a Discriminated Union.

Now we describe the storage representation for a C union variable emp as described earlier. Assuming that character occupy 1 byte, an integer 2 bytes and a float 4 bytes in memory.

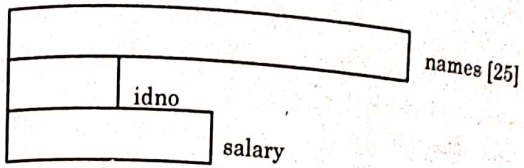


Fig. 4.19 Storage Representation for a Union in C.

Here the largest memory block requires a size of 25 bytes. So, a storage block of 25 bytes is allocated to this union variable.

4.9.1 DIFFERENCES BETWEEN STRUCTURES AND UNIONS

These are important differences between structures and unions though the syntax used for declaring them is very similar.

1. **Memory Allocation :** The amount of memory required to store a structure variable is the sum of sizes of all the members in addition to the padding bytes that may be provided by the compiler. On the other hand, in case of a union, the amount of memory required is the same as that required by its largest member. For example a union represented in figure 4.19 occupies 25 bytes memory size but the same structure occupies 31 bytes of memory size.
2. **Access of members :** While all the structure members can be accessed at any point of time, only one member of a union may be accessed at any given time. Only the last written member can be read. At this point, other variables will contain garbage.
3. **Identifying active members :** In unions, there is no way to find the active members, at any given point and therefore, the program must keep track of active members explicitly while this is implicit in records.
4. **Operations on members :** performing operations on unions as a whole, *e.g.*, arithmetic or logical operations are illegal, as in structures.

Note :-

1. The unions provide programming flexibility *e.g.* their presence in Pascal allows pointer arithmetic.
2. Unions are potentially unsafe in languages like FORTRAN, Pascal, C, C++ and Modula-2.
3. Unions can be designed so that they can be safely used as in Ada. In most other languages, unions must be used with care.

4.10 POINTERS AND PROGRAMMER – CONSTRUCTED DATA OBJECTS

Specification : The pointer type is different from the preceding primitive types. Instead of containing a data object directly, it contains the location of an object. Hence, pointer values are the memory addresses of other objects, much like the idea of indirect addressing used in assembly language. They may be called **reference** or **access** types in some languages.

For example, the memory location associated with an integer value i may contain the value 12. If p is a pointer to an integer at address 2000, then p contains the address 2000, while the location 2000 may contain an integer value 16, as shown in figure 4.20.

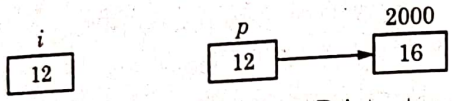


Fig. 4.20 An Integer Variable Versus a Pointer to an Integer.

Note that the pointer variable can have a special value, NIL. The value NIL is not a valid address and is used to indicate that a pointer cannot currently be pointers are used as follows :

1. **Efficiency :** Rather than move or copy a large data structure in memory, it is more efficient to move or copy a pointer to the data structure.
2. **Dynamic data :** Data structures that grow and shrink during execution can be implemented using records and pointers.
3. **Indirect Addressing :** Pointers provide some of the power of indirect addressing, which is heavily used in assembly language programming.

A single data object of pointer type might be treated in two ways :

1. Pointers may reference data objects only of a single type. This approach is used in C, Pascal and Ada ie.g *in C.*

```
int * ptr ;
```

declare a pointer ptr to an integer data object. The * designates the type of ptr as type pointer. The type **int** designates that the value of ptr may be the *l*-value of an object of type integer.

2. Pointers may reference data objects of varying types during program execution. This approach is used in the languages like small talk, where data objects carry type descriptors during execution and dynamic type checking is performed.

Operations : The languages that provide a pointer type usually includes the following operations.

1. **Dynamic allocation on heap (creation operation) :** The creation operation allocates storage for an object of the proper type, and the pointer to (address of) that object is returned. In Pascal and Ada, this is accomplished by the procedure call new (P) as shown in figure 4.21. In C, the malloc (memory allocates) provides this function *e.g.*

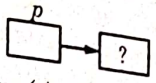


Fig. 4.21 New (*p*) Allocates Storage in Heap

p = malloc (size of (int))

Here p is a pointer to objects of type **int**. The above statement creates a two-word block of storage to be used as an object of type integer, and store its *l*-value in p.

2. **Dereferencing operation :** A dereferencing operation for pointer values allows a pointer to be followed to the data object to which it points. Dereferencing can be either implicit or explicit. In ALGOL 68 and FORTRAN 90 it is implicit, but in most other languages, it occurs only when explicitly specified. In Pascal, the expression p (in temporary) denotes the data structure pointed to by p. In C, *e.g.* $*p$ first accesses the value in p, assumes it is now an *l*-value, and uses this to access the first component of the record pointed to by p.
3. **Assignment :** The assignment operation sets a pointer variables value to some useful address. Assignments are permitted between pointers of the same type. For example, if ptr is a pointer variable with the value 2000, and the cell whose address is 2000 has the value 222, then the assignment J : = ptr ↑ sets J to 206 in Pascal as shown in figure 4.20.

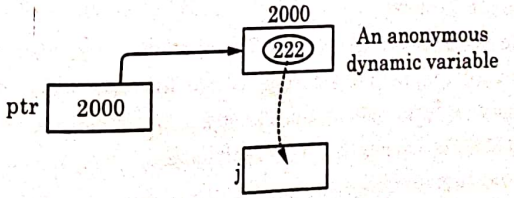


Fig. 4.22 The Assignment Operation j = ptr↑

When changing the contents of a pointer by assignment or allocation, it possible to the access to the prior stored there. This lost storage is calle **garbage**.

4. **Equality Testing :** The equality relation = tests if two pointers of t same type point to the same data structure. An inequality test ≠ is allowe as well.

5. **Deallocation :** A dynamic data structure exists until it is explicit released. Pascal provides the procedure **dispose (*p*)** to deallocate t storage at address *p*. Since several pointers may contain the same addre one must be careful not to deallocate one of them, otherwise dangli references are created. For example, if we start with the configuration figure 4.23 (*a*).

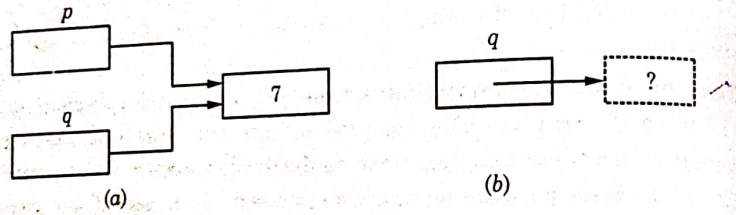


Fig. 4.23 (*a*) Initial Configuration
(*b*) **Dispose (*p*)** Creates a Dangling Reference

If we now dispose (*p*), the location 7 is stored may be reused for some oth purpose. Since q still contains this address, it is now a dangling referen into the heap as shown in figure 4.23 (*b*). The programmer must make su there are no other references to an address before deallocation.

Implementation : A pointer data object is represented as a storage locati containing the address of another storage location. The address is the base addr of the storage block representing the data object pointed to by the pointer. T major storage representations are :

1. **Absolute Address :** A pointer value may be represented as the act memory address of the storage block for the data object. The ma advantage is efficient selection. The main disadvantage with this techniq is that storage management is more difficult.
2. **Relative address :** A pointer value may be represented as an offset fr the base address of some larger heap storage block with in which the da object is allocated. Selection using this form of pointer value is more cos

than for an absolute address because the offset must be added to the base address of the area to obtain an absolute address before the data object may be accessed. However, the main advantage is in the opportunity to move the area block as a whole at any time without invalidating any of the pointers. The major implementation problems associated with pointers and programmer constructed data objects are :

1. The storage allocation associated with the creation operation.
2. Substantial extension of the overall run-time storage management structure.
3. Potential for generating garbage if all pointers to a created data object.
4. Dangling references, if the data objects can be destroyed and the storage recovered for reuse.
5. Lost heap-dynamic variables.

4.11 SETS

Specification : A set is defined as any unordered collection of distinct elements, unlike arrays, which are ordered. Set types are often used to model mathematical sets. For example, text analysis often requires that small sets of characters, such as punctuation characters or vowels, be stored and conveniently searched. An example of a set is

$$A = \{a, b, c\}$$

Here A is the **set name** and *a*, *b* and *c* are the **elements** or **members** of set A.

Among the common Imperative languages, only Pascal and Modula-2 include sets as a data type. In the Pascal model, the elements must be of homogeneous type. This base type is limited to enumeration and subrange types, since they are finite in size. The following example shows the definition of a set type and some set type variables.

```
type colors = (red, blue, green, yellow, orange, white, black) :
    colorset = set of colors ;
var set 1, set 2 : colorset ;
```

constant values can be assigned to the set variables set 1 and set 2 as in

```
set 1 : = [red, blue, yellow, white]
set 2 : = [black, blue] ;
```

Operations : The basic operations available for sets are :

1. **Membership :** The membership operation tests if an element x belongs to set A *i.e.* whether $x \in A$ or $x \notin A$. For example if

   $A = \{a, b, c\}$

   Here $a \in A$, but $d \notin A$

2. **Subset :** Given two sets A and B, A is subset of B $(A \subseteq B)$ if all the elements of A are also contained in B *i.e.*

   if $A = \{a, b, c\}$ $B = \{a, b, c, d\}$

   Here $A \subseteq B$.

   If $A \subseteq B$ and $A \neq B$ then A is proper subset of B $(A \subset B)$.

3. **Equality of sets :** Two sets A and B are equal if they have same collection of elements. Alternatively $A = B$ if $A \subseteq B$ and $B \subseteq A$ *e.g.*

   $A = \{x \mid x$ is an odd number less than $10\}$

   $B = \{1, 3, 5, 7, 9\}$

   Clearly, $A = B$.

4. **Insertion and deletion of single values :** The insertion operation inserts an element x in set A if it is not already in set A. The deletion operation delete an element x from set A.

5. **Union, Intersection and difference of sets :** Given two sets A and B.

   (*a*) **Union :** $A \cup B$ contains all the elements of A and B with duplicate elements eliminated.

   (*b*) **Intersection :** $A \cap B$ contains only the elements present in both A and B.

   (*c*) **Difference :** $A - B$ contains all the elements of A not present in B.

   (*d*) **Symmetric difference :** $A \oplus B$ contains all uncommon elements of A and B *i.e.* $A \oplus B = (A - B) \cup (B - A)$.

Implementation : The unordered sets uses the two specialized storage representations described below.

1. **Bit - string representation :** The bit string representation is useful where the limit on the maximum number of elements in the set is imposed. The purpose of the limits is to allow sets to be represented by one or more machine words. This representation is used in Pascal.

A set of n elements is implemented as a bit vector of length n. where the i^{th} bit is 1 or 0 according to whether the corresponding element is present in set or not. The bit string represents the characteristic function of the set. *e.g.* consider a set

$A = \{1, 3, 5, 9\}$

This set can be represented as 101010001 with only the 1st, 3rd, 5th and 9th bits set as 1 from the 9 bits. Using this representation.

(*a*) Insertion of an element into a set consist of setting the corresponding bit to 1.

(*b*) Deletion of an element from a set consist of setting the corresponding bit to 0.

(*c*) Membership is determined by interrogating the appropriate bit.

(*d*) Union, intersection and difference operations may be represented by Boolean operations in bit string that are usually provided by the hardware.

2. **Hash-Coded representation :** The hash-coded representation of sets is used when the underlying set contains large number of values. This representation is based on hash coding or scatter storage. To be effective hash coding methods require a substantial storage allocation. LISP implementation uses this storage representation for the set called object list, which consist of names of all atomic data objects in use by a LISP program during its execution. Almost every compiler uses hash-coding to loop up names in its symbol table.

In a vector, every subscript designates a unique component that is easily addressable with a simple accessing function. The goal with hash-coding is to duplicate this property as much as possible so that efficient accessing of a component is maintained. However, the problem is that the potential set of valid names is huge compared to the available memory.

The Hash-coded representation allows membership test, insertion and deletion of values to be performed efficiently. However, union, intersection and difference operation must be implemented as a sequence of membership tests, insertion and deletion of individual elements, so they are inefficient. As the Hash-coded representation uses hashing function but even the best hashing function cannot in general guarantee that different data items will generate different hash addresses when hashed. The two data items may be hashed to the same hash address, leading to a collision. the techniques used for handling collisions can be **rehashing, sequential scan** and **bucketing**. With a good hashing function and the table at most half full, collisions rarely occur.

4.12 FILES

A file is a collection of logically related records. Recall that a record is structure of logically related fields or elements of information. Usually all of t record occurrences on a file are of a single format, although there are also so examples of files with multiple record formats. Generally the records on a file a stored together for some common purpose ; *e.g,* they support payroll processing employee benefit record-keeping, or they contain inventory information or da gathered in a scientific experiment or a model developed using a Computer-Aide Design (CAD) system.

A file is a data structure with two special properties.

1. Files are represented on a secondary storage device such as a dish or ta and thus may be much larger than most data structures of other types.
2. The lifetime of files may encompass a greater span of time than that of th program creating it.

Reasons for file uses : There are some good reasons for structuring collection of data as a file. These are described as given below.

1. To store the data independently of the execution of a particular program.
2. For input and output of data to an external operating environment.
3. The data may be stored as a file because only a small portion of th collection is accessed by a program at any given time, making unreasonable to store the entire collection in main memory simultaneously
4. To store a data collection that is too large to fit in the program's worki memory.
5. Temporary scratch storage by data when not enough high speed memory available.

Modes of accessing Files : There are three possible modes of access to a fi by a program.

1. **Input :** An input file is only read by the program *e.g.,* a file of tax rat and tables would be an input file for the program that computes incom taxes.
2. **Output :** An output file is only written to by a program, it is created by th program. *e.g.* a report file may be the output of a program that updates master file.

3. **Input / Output File :** An input/output file is both read from and written to during a program's execution *e.g.,* the payroll master file might be used by the payroll program both as a source of data about employee pay rates and as a repository of month-to-date and year-to-date pay totals.

File Organizations : The technique used to represent and store the records on file is called the file organization. There are following fundamental file organization techniques.

1. Sequential
2. Direct - access
3. Indexed sequential

These are described in following subsections.

4.12.1 SEQUENTIAL FILES

Specification : The most basic way to organize the collection of records that form a file is to use sequential organization. In a sequentially organized file, the records are written consequently when the file is created and must be accessed consecutively when the file is later used for input as shown in figure 4.24.

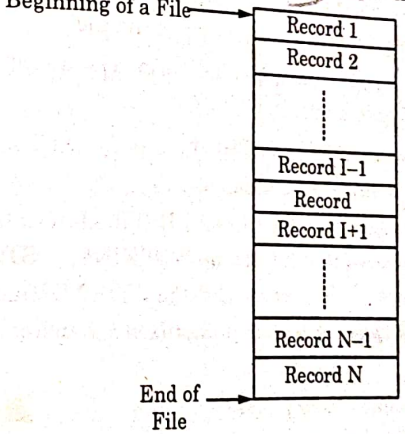


Fig. 4.24 Structure of a Sequential File

For example in Pascal,

Master : **File of** StudentRec ;

defines a file named Master whose components are of type studentRec. A sequential file may be composed of one record type or record of several different

kinds Multiple record types would be grouped together to form a file if they had common functional purpose and were closely related logically.

A distinct characteristic of a sequential file is the manner in which components (or records) may be accessed. Typically the file may be accessed in eith read mode or write mode. In either mode there is a file position pointer th designates a position either before the first file component, between two componen or after the last component.

Operations : The major operations on sequential files are :

1. **Creating a file :** The initial creation of a file is also referred to as th loading of the file. In some implementations, space is first allocated to th file, then the data are loaded into that skeleton. In other implementation the file is constructed a record at a time.
2. **Opening a file :** Before a program can access a file for input or outpu that file must be **opened**. The **open** operation is given the name of a fi and the access mode (read or write). In Pascal the procedure **reset** opens file in read mode and procedure **rewrite** open a file in write mod Sometimes a language provides for an implicit **open** operation on a file a the time of the first attempt to read or write the file.
3. **Reading a file :** Records are read form a sequential file using th following types of statements. In COBOL –

   READ filename INTO identifier

   AT END imperative – statement

   where the filename is defined in an FD (File description), the optiona INTO clause specifies an identifier in WORKING – STORAGE that wi receive the contents of the record, and the AT END clause is required t specify what is to happen when the input file is exhausted.

   In Pascal –

   read (filename, recordname) ;

   or readln (Filename, recordname) ;

   depending on the disposition of any **"left over"** characters in the inpu record, where the filename appears in the program statement and th record name variable will receive the data.

4. **Writing a file :** A write operation creates a new component at the content position in the file (always at the end) and transfers the contents of a designated program variable to the new component. In COBOL.

   WRITE record – name [FROM identifier]

   where the record name is defined in the file's FD (file description) and the optional identifier of the FROM clause is defined in WORKING – STORAGE.

   In Pascal –

   writeln (Filename, recordname) ;

   or write (Filename, record name) ;

   depending upon whether or not a new line is to be started after this one, where the filename appears in the program statement and the record name variable contains the data to be written. The record will appear on the file in the same order as they are written.
5. **Updating a file :** Changing the contents of a master file to make it reflect a more current snapshot of the real world is known as **updating** the file. These change may include the insertion, deletion and modification of records.
6. **Retrieving from a file :** The access of a file for purpose of extracting meaningful information is called **retrieval**. There are two basic classes of file retrieval inquiry and report generation.
7. **Maintaining a file :** Changes that are made to files to improve the performance of the programs that access them are known as **maintenance** activities. There are two basic classes of maintenance operations **restructing** and **re-organization**.
8. **End-of-file test :** An explicit test for the end-of-file position is needed so that the program may take special action Pascal provides a function ;

   eof : file → Boolean

   that returns true if the file is positioned at its end, and false otherwise.
9. **Closing a file :** After a program has completed its use of a file, the file needs to be closed. This processing may be initiated by a CLOSE statement or by default at program termination. Closing a file prepares it for later use by another program execution.

Implementation : The primary responsibility for file implementation is that of underlying operating system while a particular programming language implementation provides only the necessary data structures needed for the interface

with the operating system. File operations are primarily implemented by calls on primitives provided by the operating system. When a program opens a file during its execution, storage for a file information table and a buffer must be provided. The buffer is used as a sequential representation of a queue. When a **write** operation is performed, the block of components is transferred from the buffer to the external storage device (*e.g.* disk or tape). When a **read** operation is performed the data is transferred. From the file into the buffer in blocks of components. The cost, for the program of a read or write operation is reduced to the cost of copying a component from program variable into the buffer or vice-versa, which is fast and inexpensive compared to the speed and cost of the actual transfer of a block to the file. This organization is shown in figure 4.25.

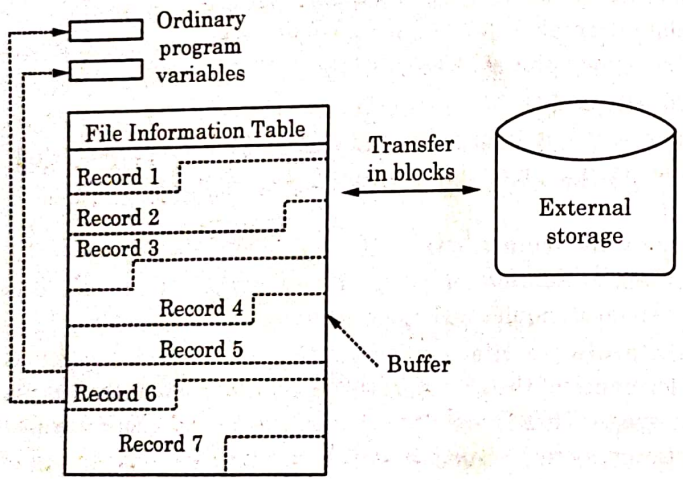


Fig. 4.25 File Representation Using a Buffer.

4.12.2 DIRECT-ACCESS FILES

An effective way to organize a file when there is a need to access individual records directly is direct-access file organization. *i.e.* in direct-access files the components may be selected at random. The subscript used to select a component is called its key. There is predictable relationship between the key used to identify a particular record and that record's location in the file. However, it is important to understand that the logical ordering of records need bear no relationship to their physical sequencing as shown in figure 4.26. The records do not necessarily appear physically in sorted order by their key values.

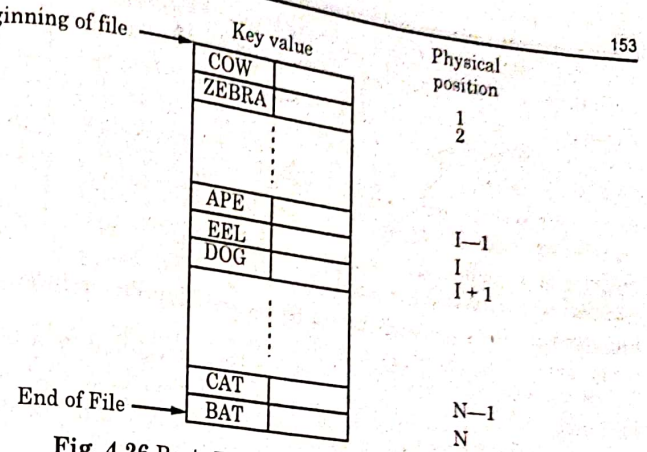


Fig. 4.26 Basic Direct-Access File Organization.

In order to access N^{th} record, the relationship used to translate between the key values and the physical addresses is designated by a mapping function.

R(key value) → address

from key values to addresses in the file.

4.12.3 INDEXED SEQUENTIAL FILES

An effective way to organize a collection of records when there is the need both to access the records sequentially by some key value and also to access the records individually by that same key is **indexed sequential file organization.** An indexed sequential file provides the combination of access types that are supported by a sequential file and a direct access file. One way to think of the structure of an indexed sequential file is as an index with pointers to a sequential data file as shown in figure 4.27.

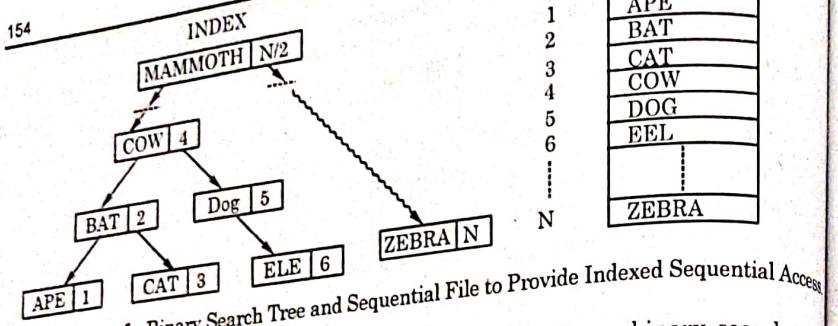


Fig. 4.27 Use of a Binary Search Tree and Sequential File to Provide Indexed Sequential Access

In the example above, the index has been structured as a binary search tree. The index is used to service a request for access to a particular record ; the sequential data file is used to support sequential access to the entire collection of records.

KEY POINTS TO REMEMBER

- A **data object** refers to run-time grouping of one or more pieces of data in a virtual computer.
- A **structured data object (or data structure)** is composed of other data objects)
- A **structured data object** can be either **system defined** or **programmer defined.**
- The main attributes of the structured data types are :
  - The number of components
  - The type of each component
  - Names of be used for selecting components
  - Maximum number of components
  - Organization of the components.
- Based on the number of components, the data structure can be either of **fixed size** or **variable size.**
- Based on the type of each component, the data structure can be either homogenous or heterogeneous.

- The operations on data structures are :
  - Component selection operations
  - Whole data – structure operations
  - Insertion /deletion of components
  - Creation/deletion of data structures
- The basic elements of structure data types implementation are :
  - Storage Representation
  - Implementation of operations
  - Storage management and data structure
- The storage can have **sequential** or **linked** representation.
- The declaration of structured data objects provides the information regarding the name and types of data structure.
- The **type checking** for structure data objects is more complex transfer elementary data objects.
- An **array** is a homogenous data structure in which all the elements are of same type.
- An array can be one, two or multidimensional.
- In case of the implementation of one- dimensional array (vectors) we have

$$l\text{ value}(A[I]) = L + (I-1)\times E$$ where

L = The location at which the first elements of the vector begins and LB = 1, E = size of each components

I = Component to be addressed

- The two dimensional arrays can be represented in storage using :
  - Row -major order , or
  - Column major order.
- The **access formula** to computes the offset of a components in a two dimensional array is given by

$$l\text{-value}(A[I, J]) = L + (I - (B_1)\times S(J - LB_2)\times E$$

where L = Base address, S = Length of row = $(UB_2 - LB_2 + 1)\times E$

LB_1, LB_2 = Lower Bounds on first and second subscript

UB_1, UB_2 = Upper Bounds on first and second subscript

- The address of array element in multidimensional array $A[S_1, S_2, \ldots\ldots, S_n]$ is given by

$$l \text{ value } (A[S_1, S_2, S_n]) = V_0 = \sum_{i=1}^{n} (S_i \times m_i)$$

where V_0 = virtual origin $= \alpha - \sum_{i=1}^{n} (L_i \times m_i)$

- A **slice** is a substructure of an array that it itself on array.
- A **record** is a possibly heterogeneous aggregate of data elements in which the individual elements are identified by names.
- The **variant records** are used for representing objects that have some but not all properties in common.
- A **string** is defined as a finite sequence of alphanumeric characters.
- The strings can have
  - Static (fixed). string length
  - Variable length scheme with a fixed maximum
  - dynamic (unbounded) string length
- The various operations that can be applied on strings are concatenation, pattern matching, subring selection, Rerational operations, and Input-output formatting.
- A **list** is a data structure composed of an ordered sequence of data structures.
- The several variations of lists stacks, queue, tree, directed graph, and property lists.
- A **property** list is defined as the record with varying number of components if the components may vary without restriction.
- A **union** is a type that may store different type values at different times during the program execution.
- The pointer types contain the location of a data object. They may be called **reference** or **access** types in some languages.
- A **set** is defined as any unordered collection of distinct elements.
- The basic operation can sets includes membership, subset, equality of sets, insertion and deletion of single values, union, intersection and difference of sets.

- The unordered sets uses two specialized storage representations.
  - Bit string representation
  - Hash-coded representation
- A **file** is a collection of logically related records.
- The three basic file organization techniques are
  - Sequential files
  - Direct-access files
  - Index-sequential files.

EXERICISE

1. Explain the terms structured data object and structured data type in detail.
2. Explain the specification and implementation of structure data types in detail.
3. Explain the row major order and column major order implementation of 2-D arrays.
4. What are character strings ? Explain different concepts related to character string in detail.
5. What are lists. What are different variations on lists ?
6. What are files ? Explain different file organisation techniques.
7. Write short notes on :
   (*a*) Vectors and arrays
   (*b*) Pointers and programmer constructed data objects
   (*c*) Records and variant records
   (*d*) Structures and unions

CHAPTER

5 SUBPROGRAMS AND PROGRAMMER DEFINED DATA TYPES

5.1 INTRODUCTION

Each programming language provides some built in data types so that programmers need not to be concerned with the details of the underlying representation of data type as bit sequences. For example, C language provides integer and float of built in data types. In the construction of large programs, the programmer is generally concerned with the design and implementation of new data types. A set of operations are also required to be defined to provide the basic manipulations. A particular representation for new data types must be chosen, using the data types provided by the language or perhaps using other abstract data types.

The current goal in programming language design is to make the distinctions between the various forms of data transparent to the programmer who uses the data types. A programmer who needs a built-in-data type (*e.g.* integer) and one who needs a user defined data type should see the same syntax. Similar, the corresponding operation signatures should have the same syntax and comparable semantics.

The three basic mechanisms exist to provide the programmer with the ability to create new data types and operations in that type.

1. **Subprograms,** created by programmers to perform functionality of new type.
2. **Type declaration,** to define new types and operations on that type e.g. abstruct data types in C, Pascal and Aa.
3. **Inheritance,** an object oriented programming concept to expand the ability of the programmer to define new types and operations on those types and detect improper uses of that type.

5.2 EVOLUTION OF THE DATA TYPE CONCEPT

Computer programs produce results by manipulating data. An important factor in determining the ease with which they can perform this task is how well the data

types match the real-world problem space. It is therefore crucial that a language support an appropriate variety of data types and structures.

The contemporary concepts of data typing have evolved over the last 40 years. The beginning of data type concept is generally seen in the older languages like FORTRAN and COBOL. In the earliest languages, all problem space data structures had to be modeled with only a few language supported data structures. For example, in pre-90 FORTRANs, linked lists and binary trees are commonly modeled with arrays.

The data structure of COBOL took the first step away from the FORTRAN I model by allowing programmers to specify the accuracy of decimal data values, and also by providing a structured data type for records of information. PL/I extended the capability of accuracy specification to integer and floating point types.

However, the actual computers include no provision for defining and enforcing data type restrictions. The underlying computer hardware cannot differentiate between a bit string representing an integer, character or real number. The higher-level languages provide a set of basic data types, such an integer real and character string. Type checking is provided to ensure that the operations (*e.g.* addition, multiplication etc receives proper number of arguments of proper data types. The types of variables (data objects) are provided by means of declarations. A declaration provides a name for a data object and defines its type. The early notion of data type therefore defines a type as a set of **values** that a data object might take on.

The evolution of the data type concept was a major development in programming languages in 1970s. In 1970s the concept of data type was extended to a general type definition applicable to a set of variables. A **type definition** defines the structure of a data object with its possible value bindings. To get a particular data object of the defined type, a declaration requires only the variable name and the name of the type to be given.

The concept of the data type was further extended to include the set of operations for manipulating those data objects. For primitive types (*e.g.* integer, real etc), a programming language provides a facility to declare variables of that type and a set of operations on primitive types that represent the only way that primitive types can be manipulated by the programmer. Thus the storage representation of primitive types is hidden from the programmer (*i.e.* **encapsulated**). The programmer may use the primitive data types without knowing their storage representation details.

A better approach, introduced in ALGOL 68, is to provide a few basic types and a few flexible structure – defining operations that allow a programmer to tailor a structure to the problem at hand with user-defined data types. This was clearly one of the most important advances in the evolution of data type design. User defined types provide improved readability through the use of meaningful names for types. They allow type checking of the variables of a special category of use, which would otherwise not be possible. User defined types also aid **modifiability**. A programmer can change the type of a category of variables in a program by changing only a type declaration statement.

Ada 83 embodied the contemporary concepts in data type design of the late 1970s, which resulted from a natural extension to the idea of user-defined types. The philosophy of user-defined data types is that the user should be allowed to create a unique type for each unique class of variables in the problem space. Moreover, the language must enforce the uniqueness of the types, which are in fact abstractors of the problem space variables. This is a powerful concept, and it has a significant impact on the overall process of software design. Taking this concept a step farther, we arrive at **abstract data types**, which can be simulated in Ada 83. The fundamental idea of an abstract data type is that the use of a type is separated from the representation and set of operations on values of that type. All of the types provided by a high-level programming language are abstract data types.

5.3 ABSTRACTION, ENCAPSULATION AND INFORMATION HIDING

To understand the design of language facilities for programmer defined data types and operation, it is important to understand the basic concepts of **abstraction**, **encapsulation** and **information hiding**. These concepts are discussed in following subsections.

5.3.1 ABSTRACTION

The first thing with which one is confronted when writing programs is the problem. Typically you are confronted with "real-life" problems and you want to make life easier by providing a program for the problem. However, real-life problems are nebulous. A large program, when all its details are considered once, easily exceeds the intellectual grasp of a single person. Therefore, in order to understand problem it is necessary to separate the necessary details from unnecessary tails. You try to obtain your own abstract view or model of the problem. This process of modelling is called **abstraction** as illustrated in figure 5.1.

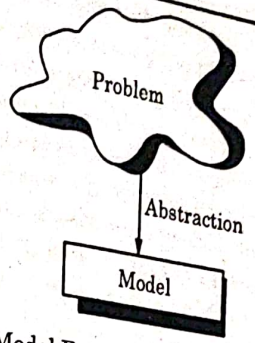


Fig. 5.1 : A Model From a problem with Abstraction.

The model defines an abstract view to the problem. This implies that the model focuses only on problem related stuff and that you try to define properties of the problem. These properties includes :

1. The data which are affected, and
2. The operations which are identified.

The above discussion infer the fact that abstraction is an emphasis on the idea, qualities and properties rather than the particulars (a suppression of detail). An abstraction is s view on representation of an entity that includes only the attributes of significance in a particular context.

Abstraction allows one to collect instances of entities into groups in which their common attributes need not be considered. These common attributes are abstracted away. With in the groups, only the attributes that distinguish the individual elements need be considered. Abstraction is a weapon against the complexity of programming ; its purpose is to simplify the programming process. It is an effective weapon because it allows programmers to focus on essential attribute and ignore subordinate attributes.

As an example consider the administration of employees in an institution. The need of the administration comes to you and ask you to create a program which allows to administer the employees. Well, this is not very specific. The employees are real persons who can be characterized with many properties *e.g.* name, date_of_birth, social number, department, address, hobbies etc. Certainly not all of these properties are necessary to solve the administration problem. Only some of them are problem specific. Consequently you create a model of an employee for the problem. This model only implies properties which are needed to fulfill the requirements of the administration. For instance name, date_of_birth and social

number. These properties are called the **data** of the (employee) model. Now you have described real persons with the help of an abstract employee.

Of course, the pure description is not enough. There must be some operations defined with which the administration is able to handle the abstract employees. For example, there must be an operation which allows you to create a new employee once a new person enter the institution. Consequently, you have to identify the operations which should be able to be performed on an abstract employees. You also decide to allow access to the employees data only with associated operations. This allows you to ensure that data elements are always in a proper state. For example you are able to check if a provided date is valid.

To sum up, abstraction is the structuring of a nebulous problem into well-defined entities by defining their data and operations. Consequently, these entities combine data and operations. They are not decoupled from each other. The importance of abstraction is derived from its ability to hide irrelevant details and from the use of names to reference objects. A programming language provides support for abstraction in following ways :

1. By providing a virtual computer that is simpler to use and more powerful than the actual underlying hardware computer the language directly supplies a useful set of abstractions as the "**features**" of the language.
2. The programming languages provide abstractions through subprograms, subprogram libraries, type definitions, classes and packages which permit the programmer to distinguish between **what** a program does and **how** it is implemented. The two fundamental kinds of abstraction in contemporary programming languages are process abstraction and data abstraction. These are described in subsequent sections.

5.3.1.1 PROCESS ABSTRACTION

The concept of process abstraction among the oldest in programming language design. The process abstraction specifies the action of a computation in a set of input objects and the output objects (s) produced. A process (or procedural) for simplifying a program is achieved by specifying a process or function (subprogram) to be performed. A subprogram would have its own name and might contain declarations of procedures and functions. A language may even support separate compilation of some types of subprograms. A program will generally have the following sections :

(1) Input data,
(2) Process data,
(3) Output results

The program could be decomposed into three parts, one responsible for each of the three activities. These subprograms are process abstractions because they provide a way for a program to specify that some process is to be done, without providing the details of how it is to be done (at least in the calling program). A subprogram module could include abstract data types as well as other functions and procedures. We can think of such a subprogram module as a **"black box"**. Known inputs enter the box, and verifiable results come out. The details of what goes on in the box, however, are hidden. For example, when a program needs to sort an array of numeric data objects of some type, it usually uses a subprogram for the sorting process. At the point where the sorting process is required, a statement such as

```
sort_int (list, list_len)
```

is placed in the program. This call is an abstraction of the actual sorting process, whose algorithm is not specified. The call is independent of the algorithm implemented in the called subprogram.

In the case of the subprogram sort_int, the only essential attributes are the name of the array to be sorted, the type of its elements, the array's length, and the fact that the call to sort_int will result in the array being sorted. The particular algorithm that sort_int implements is an attribute that is not essential to the user.

Process abstraction is crucial to the programming process. The ability to abstract away many of the details of algorithms in subprograms makes it possible to construct, read and understand large programs. All subprograms, including concurrent subprograms and exception handlers are process abstractions.

5.3.1.2 DATA ABSTRACTION

The term **"data"** is something like "raw material for a computer". **Data** may be defined as a collection of facts used as a basis for inference. The high level programming languages look at data according to what can be done to do and with it.

The evolution of data abstraction necessarily followed that of process abstraction, because an integral and central part of every data abstraction is its operations, which are defined as **process abstractions.**

A **data abstraction** consists of a set of objects and a set of operations characterizing their behaviour. For each sort of data, certain operations apply either to select out parts or to put parts together. For example, if our data is composed of names, *i.e.* character strings, one selector might print out the last name of a string. A constructor could, when combined with a selector append an appropriate address to a name, or could produce a list of all names where the last name begins with A.

What is important is to remember is that only certain selectors and construct apply to certain types of data. It makes no sense to multiply two names together construct a single object from two others, or to select out the first name of an integ

To sum up data abstraction can be defined briefly as the pair [objec operations]. One of the motivation for data abstraction is similar to that of proce abstraction. It is a weapon against complexity, a means of making large and complicated programs more manageable. The availability of data abstraction allow very different program design methodology. In the last few years, a ne methodology for software development has become increasingly popular – obje **oriented programming**. The object oriented programming is an outgrowth of th use of the data abstraction in software development, and data abstraction is one its most important components.

5.3.2 INFORMATION HIDING

Information hiding is the term used for the central principle in the design programmer –defined abstraction. In computer science, the principle of **informatio hiding** is the hiding of design decisions in a computer program that are most like to change, thus protecting other parts of the program from change if the desig decision is changed. The protection involves providing a stable interface whi shields the remainder of the program from the implementation (the details that a most likely to change).

In modern programming languages the principle of information hiding is make visible all that is essential for the user to know and to hide everything els Each program component should hide as much information as possible from th users. Thus, the language provided square-root function is a successful abstra operation because it hides the details of the number representation and the square root computation algorithm from the user.

Similarly, a programmer –defined data type is a successful abstraction if it m be used without knowledge of the representation of objects of that type or of th algorithms used by its operations. When information is encapsulate decisi abstraction, it means that the user of the abstraction.

1. Does not need to know the hidden information. The hidden complexities the hardware, may abstract the user from making simple and e algorithms.
2. Is not allowed to directly use or manipulate the hidden informati Sometimes user is not authorized to access to manipulate the hidd information. So, encapsulation is needed.

For example, the integer data type in programming language such as FORTRAN or C not only hides the details of the integer a number representation but also effectively encapsulates the representation so that the programmer cannot manipulate individual bits of the representation of an integer.

The most common uses of information hiding are

1. To hide the physical storage layout for data so that if the physical storage layout is changed, the change is restricted to a small subset of the total program.
2. To make a program easier for the user to understand.
3. To make program portable between different languages and machines.
4. To make certain security measures practical.
5. To make visible necessary and essential details and to hide everything else.

In object–oriented programming, information hiding reduces software development risk by shifting the code's **dependency** on an uncertain implementation (design decision) onto a well-defined interface so if the implementation changes, the clients do not have to change. The information hiding facilities following two kinds of changes in object-oriented programming languages.

1. **Implementation changes :** If all interactions with an object are through its interface, then algorithms and data structures hidden behind the interface can be changed.
2. **Inheritance changes :** If all interactions are through the interface to a superclass, then the program can be extended by adding subclasses.

In modern programming languages, the principle of information hiding manifests itself in a number of ways, including encapsulation (given the separation of concerns) and polymorphism. The concept of encapsulation is discussed in next section.

5.3.3 ENCAPSULATION

Encapsulation is the method of information hiding or abstraction, in which data set of information, neither can be viewed by user nor the user can manipulate or access the hidden information. Therefore, the technique of hiding the used data structure and to only provide a well-defined interface is known as **encapsulation.**

Need for Encapsulation : When the size of a program reaches beyond a few thousand lines, two practical problems appear.

1. From the programmer's point of view, having such a program appear as a single collection of subprograms does not impose an adequate level of organization on the program to keep it intellectually manageable.
2. Another practical problem for larger programs is recompilation. In the case of a small program recompiling the whole program after each modification is not costly. But when programs grow beyond a few thousand lines, the cost of recompilation ceases to be significant.

The solution the first problem is to organize the program into a syntactic containers that include group of logically related subprograms and data. These syntactic containers are often called **modules** and the process of designing them is called **modularization**. The module design has generally follows two approaches :

(*a*) Modules represent a functional decomposition of the program.

(*b*) Modules represent a data decomposition of the program.

The solution to the second problem can be provided by organizing programs into collections of subprogram and data, each of which can be compiled without recompilation of the rest of the program. Such a collection is called a **compilation unit.**

An **encapsulation** is a grouping of subprograms and the data they manipulate. An encapsulation, which is either separately or independently compliable, provides an abstracted system and a logical organization. For a collection of related computation. Therefore, an encapsulation solves both of the practical problems described above. Encapsulation are often placed in libraries and made available for sense in programs other than those for which they are written.

ENCAPSULATION AND INFORMATION HIDING

The term **encapsulation** is often used interchangeably with information hiding, while some make distinctions between the two. It seems that people however fail to agree on the distinctions between information hiding and encapsulation though one can think of following distinctions between the two.

1. An encapsulation is a compilation unit that can include a collection of logically related types, objects and subprograms. Often that definition is misconstructed to mean that the data is somehow hidden. In Java, you can have encapsulated data that is not hidden. However, hiding data is not the full extent of information hiding.
2. Information hiding can be though of as being the principle and encapsulation being the technique.

3. Information hiding is a question of program design ; information hiding is possible in any program that is properly designed, regardless of the programming language used. Encapsulation, however, is primarily a question of language design ; an abstraction is effectively encapsulated only when the language prohibits access to the information hidden with in the abstraction.

Advantages of Encapsulation :

There are following advantages of encapsulation :–

1. Encapsulation is particularly important in permitting easy modification of a program.
2. An encapsulation may also provide access control to its entities.
3. Encapsulation provide the programmer with a method of organizing programs that also limits recompilation.

The languages that provide the encapsulation facilities include SIMULA 67, Ada and C + +.

5.4 ABSTRACT DATA TYPES

An **abstract data type (ADT)**, simply put, is an encapsulation that indicates only the data representation of one specific data types and the subprograms that provide the operations for that type. Through access control unnecessary details of the type can be hidden from units outside the encapsulation that use the type.

In computing an abstract data type (ADT) is a specification of a set of data and the set of operations that can be performed on the data. Such a **data type** is abstract in the sense that it is independent of various concrete implementations.

Many programming languages defined an abstract data type (ADT) as a set of data values and associated operations that are precisely specified independent of any particular implementation. As ADTs provide an abstract view tp describe properties of set of entities, their use is independent from a particular programming language. Each ADT description consists of two parts :

1. **Data :** This part describes the structure of the data used in the ADT in an informal way.
2. **Operations :** This part describes valid operations for this ADT, hence, it describes its interface. We use the special operation **constructor** to describe the actions which are to be performed once an entity of this ADT is created and **destructor** to describe the actions which are to be performed once an entity of this ADT is created and **destructor** to describe the actions

which are to be performed once an entity is destroyed. For each operation the provided arguments as well as preconditions and post conditions are given.

5.4.1 PROPERTIES AND EXAMPLE OF ADTs

The employee administration example of section 5.3.1 shows, that with abstraction you create a well-defined entity which can be properly handled. These entities define the **data structure** of a set of items. For example, each administered employee has a name, date of birth and social number.

The data structure can only be accessed with defined **operations.** This set of operations is called **interface** and is **exported** by the entity. An entity with the properties just described is called an abstract data type (ADT).

Figure 5.2 shows an ADT which consists of an abstract data structure and operations. Only the operations are viewable from the outside and define the interface.

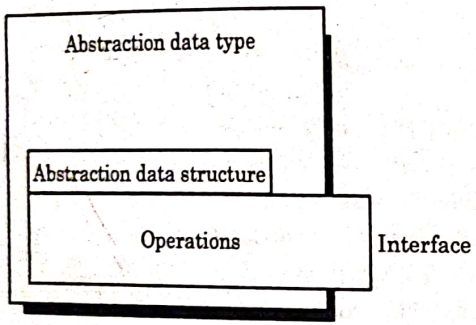


Fig. 5.2 An Abstract Data Type (ADT).

Once a new employee is "created" the data structure is filled with actual values. You now have an instance of an abstract employees. You can create as many instances of an abstract employee as needed to describe every real employed person.

Let's try to put the characteristics of an ADT in a more formal way :

Definition (Abstract Data Type) : An abstract data type (ADT) is characterized by the following properties :

1. It exports a **type.**
2. It exports a **set of operations.** This set is called interface.
3. Operations of the interface are the one and only access mechanism to the type's data structure.
4. Axioms and preconditions define the application domain of the type.

5.4.2 ABSTRACT DATA TYPES (ADTs) AND OBJECT ORIENTATION

Abstract data types (ADTs) are one of the primary components of object oriented programming. In object-orientation ADTs are referred to as classes. ADTs allow the creation of instances (called objects) with well defined properties and behaviour. Therefore, a class defined properties of objects which are the instances in an object – oriented environment. ADTs define functionality by putting the main emphasis on the involved data, their structure, operations as well as axioms and pre-conditions. Consequently, object-oriented programming is "programming with ADTs". Combining functionality of different ADTs to solve a problem. Therefore instances (objects) of ADTs (classes) are dynamically created, destroyed and used.

5.4.3 FLOATING-POINT AS AN ABSTRACT DATA TYPE

The concept of abstract data types, atleast in terms of built-in types, is not a recent development. Most languages include a floating point data type, which provides a means of creating variables for floating-point data and also provides a set of arithmetic operations for manipulating objects of the type.

Floating-point types in high-level languages employ a key concept in data abstraction : information hiding. The actual format of the data value in a floating – point memory cell is hidden from the user. The only operations available are those provided by the language. The user is not allowed to create new operations on data of the type, except those that can be constructed using the built-in operations. The user cannot directly manipulate the parts of the actual representation of floating point objects because that representation is hidden. It is this feature that allows program portability between implementations of a particular language, even though the implementation may use different representations of floating point values.

5.4.4 USER-DEFINED ABSTRACT DATA TYPES

The concept of user-defined abstract data types is a relatively recent one. Like Built-in types, a user defined ADT should provide the following characteristics.

(*a*) A type definition that allow program units to declare variables of the type but hides the representation of these variables, and

(*b*) A set of operations for manipulating objects of the type.

An ADT in the context of user-defined data types is a data type satisfying following two conditions :

(1) The representation, or definition, of the type and the operations on objects of the type are contained in a single syntactic unit. Also, other program units may be allowed to create variable of the defined type.

(2) The representation of objects of the type is hidden from the program units that use the type, so the only direct operations possible on those objects are those provided in the type definition.

Program units that use a specific abstract data type are called **clients** of that types.

5.4.5 ADVANTAGES

There are following advantages of ADTs.

1. The packaging of representation and operations in a single syntactic unit provides the advantages of encapsulation.
2. It provides a method of organizing a program into logical units that can be compiled separately.
3. The modifications on the representations or operations of the type to be done in a single area of the program.
4. The clients are not able to "see" the representation details, and thus their code cannot depend on that representation.
5. The representation can be changed at any time without requiring changes to the clients.
6. It increases reliability.
7. A great deal of recompilation can be avoided in the software development process.

5.5 SUBPROGRAMS

Subprograms are the fundamental building blocks of programs and are therefore among the most important concepts in programming language design. A subprogram is defined as a collection of statements that can be reused at several different places in a program when convenient. This reuse results in several different kinds of savings, from memory space to coding time. Such reuse is also an abstraction, for the details of the subprogram's computation are replaced in a program by a statement that calls the subprogram. Instead of explaining how some computation is to be done in a program, that explanation (the collection of statements in the subprogram) is enacted by a call statement, effectively abstracting

away the details. This increases the readability of a program by exposing its logical structure while hiding the small scale details.

5.5.1 SUBPROGRAM CHARACTERISTICS

The fundamental characteristics of subprograms are :

(1) A subprogram has a single entry point.

(2) The caller is suspended during execution of the called subprogram.

(3) Control always returns to the caller when the called subprogram's execution terminates.

5.5.2 PROCEDURES AND FUNCTIONS

There are two distinct categories of subprograms : **procedures** and **functions**, both of which can be viewed as approaches to extending the language.

Procedures : A **procedure** is defined as a subprogram that define parameterized computations. These computations are enacted by single call statements. In effect, procedures define new statement. For example, because Pascal does not have a sort statement, a user can build a procedure to sort arrays of data and use a call to that procedure in place of the unavailable sort statement.

The general syntax of a procedure in Pascal is given as

```
PROCEDURE  Name of Procedure (formal parameter list) ;
    { local declaration section }
BEGIN
    { instruction sequence }
END ;
    { end of procedure }
```

The above declaration implies that a procedure has two parts : The **specification** and the **body**. The procedure specification begins with the keyword PROCEDURE and end with the procedure name or a parameter list. Parameter declarations are optional. Procedures that take no parameters are written without parentheses.

The procedure body begins with the keyword BEGIN and end with the keyword END followed by an optional procedure name. The procedure body has three parts : a declarative part, an executable art and an optional exception handling part.

Procedures can produce results in the calling program by two methods :

1. If there are variables that are not formal parameters but are still visible in both the procedure and the calling program unit, the procedure can change them.
2. If the subprogram has formal parameters that allow the transfer of data to the caller, those parameters can be changed.

In general the procedures produce side effects *i.e.* it modifies the parameters and any variables defined outside the procedures.

Functions : A **function** is a subprogram that computes a value. Function and procedures are structured alike, except that

(*a*) Functions are semantically modeled on mathematical functions.

(*b*) Functions have a RETURN clause.

(*c*) Functions produces no side effects, *i.e.* it modifies neither its parameters nor any variables defined outside the function.

The general syntax of a function in C is given as

```
RETURN TYPE    Name of Function (formal parameter list)
    {
        local declaration section
        .....
        .....
        instruction sequence
    }
```

Like a procedure, a function has two part : the **specification** and the **body**. The function specification begins with the Return type followed by name of function and parameter list. The function body begins with { and ends with }. The function body has three parts : a **declaration** part, **an executable** part, and an optional **exception-handling** part.

The functions are called by appearances of their names in expressions, along with the required actual parameters. The value produced by a function's execution is returned to the calling code, effectively replacing the cell itself. Most common imperative languages provide both functions and procedures. C and C ++ have only functions. However, these functions can behave like procedures. They can be defined to return no value by defining its return type to be void.

5.5.3 ADVANTAGES OF SUBPROGRAMS

The subprograms provide the following advantages in programming languages :–

1. Subprogram promote reusability : *i.e.* to reuse a collection of statements at several different places in a program.
2. Subprograms provide extensibility *i.e.* they let you tailor the language to suit your needs *e.g.*, if you need a procedure that creates new departments, you can easily write one.
3. Subprograms provide modularity *i.e.* they let you break a program down into manageable, well-defined logic modules. This supports top-down design and the stepwise refinement approach to problem solving.
4. Subprogram promote maintainability *i.e.* if its definition changes, only the subprogram is a affected. This simplifies maintenance and enhancement.
5. Subprograms aid abstraction, *i.e.* to use subprograms, you must know what they do, not how they work. Therefore, you can design applications from the top down without worrying about implementation details.

5.6 ENCAPSULATION BY SUBPROGRAMS

The two important views of subprograms are :

1. At the program design level, a subprogram represents an abstract operation that the programmer defines, as opposed to the primitive operations built into the language.
2. At the language design level, the concern is with the design and implementation of general facilities, for subprogram definition and invocation.

These two views of subprograms are discussed in next two subsections.

5.6.1 SUBPROGRAMS AS ABSTRACT OPERATIONS

As described earlier, a subprogram (procedure and function) has two parts : a specification and implementation. These are discussed as given below :

Specification : As a subprogram represents an abstract operation, we should be able to know what they do (specification) not how they work (implementation). The specification of a subprogram includes :

1. The **name** of the subprogram.
2. The **signature** (or **prototype**) of the subprogram giving the number of arguments, their order, and the data type of each, and the number of results, their order, and data type of each. *e.g.* consider a subprogram (function) specification in C :

float emp (int a, float b)

which specifies the signature (prototype) as

emp : int × float → float

3. The action performed by the subprogram.

The two fundamental kinds of subprograms in various programming languages are : procedures and functions as described in section 5.5.2.

While a subprogram represents a mathematical function, problems arise in attempting to describe precisely the function computed :

1. A subprogram may have implicit arguments and implicit results (side effects).
2. A subprogram may not be defined for some possible arguments.
3. A subprogram may be history sensitive.

Implementation : The implementation of a subprogram is defined by the subprogram body which consists of local data declarations and statements defining the actions. The declarations and statements are usually encapsulated so that neither the local data nor the statements are accessible separately to the user of the subprogram ; the user may only invoke the subprogram with a particular set of arguments and receive the computed results. The (syntax for the body of a subprogram is given as)

```
int emp (int a, float b) → signature
{
    int x;          } → local data Declarations
    float z;
    .....
    .....
}                   } → sequence of statements defines actions
```

Some programming languages like Pascal and Ada allow definition of other subprograms to be included with in the body of subprogram.

5.6.2 SUBPROGRAM DEFINITION AND INVOCATION

Specification : A subprogram definition describes the interface to and the actions of the subprogram abstraction. A subprogram invocation (subprogram call) is the explicit request that the subprogram be executed. A subprogram is said to be active if, after having been called, it has begun execution but has not yet completed that execution.

A programmer writes a subprogram definition as a static property of a program. During execution of the program, if the subprogram is invoked an activation of the subprogram is created. When execution of the subprogram is complete, the activation is destroyed. If another call is made, a new activation is created. From a single subprogram definition, many activations may be created during program execution. The definition serves as a template for creating activations during execution. The differences between subprogram definition and activation are summarized in table 5.1.

Subprogram Definition	Subprogram Activation
1. It describes on the interface to and the actions of the subprogram abstraction.	1. The subprogram activation is constructed from its template provide subprogram definition.
2. The subprogram definition is available at compile time.	2. The subprogram activation exists only during the program execution.
3. During translation, the type of subprogram is known but not their value or location.	3. During execution, code to access a variables value or location can be executed but the type may not be available.

Table 5.1 Differences between the subprogram definition and subprogram activation.

Implementation : The subprogram definition defines the following components needed for an activation of the subprogram at run time.

1. A signature line providing information for storage for parameters and function results.
2. Declarations providing storage for local variables.
3. Storage for literals and defined constants.
4. Storage for executable code.

The above components are shown in following subprogram definition syntax

```
RETURN TYPE subprograminame (formal parameter list)
{
    Declarations
    literals and constants
    .....
    executable code
    .....
}
```

The subprogram definition allows the above mentioned storage areas to b organized and executable code determined during translation. The result fro translation is the template used to construct each particular activation at the tim the subprogram is called during execution.

To construct a particular activation of the subprogram from its template, th entire template might be copied into a new memory area. However, rather tha making a complete copy, it is far better to split the template into two parts :

1. **Code Segment :** The code segment is the static part, consisting of th constants and executable code. A single code segment exists in storage throughou the program execution, *i.e.* code segment should be invariant during execution o the subprogram and therefore a single copy may be shared by all activations. Th code segment for a subprogram is as shown in figure 5.3.

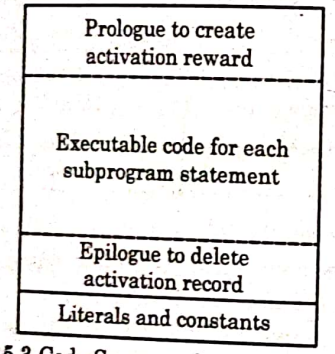


Fig. 5.3 Code Segment for a Subprogram.

When a subprogram is called, a number of hidden actions takes place. There are basically two such actions as shown in figure 5.3.

(*a*) **Prologue :** A prologue is a set of instructions inserted by the translators at the start of subprogram for setting up of the activation record, the transmission of parameters, the creation of linkages for non local referencing and similar **"housekeeping"** activities. These actions must take place before the actual code for the statements in the body of the subprogram is executed.

(*b*) **Epilogue :** An epilogue is a set of instructions inserted by the translator at the end of the executable code block to perform actions like to return results and free the storage. For the activation record and similar **"housekeeping"** activities.

2. **Activation record :** The format, or layout, of the noncode part of a subprogram is called an activation record, because the data it describes are only relevant during the activation of the subprogram. As activation record is a dynamic part consisting of parameters, function results and local data and other implementation – defined **"housekeeping"** data such as temporary storage areas, return points and linkage. For referencing non local variables. This part has the same structure for each activation, but it contains different data values. Thus each activation necessarily has its own copy of the activation record part. The activation record for a subprogram is as shown in figure 5.4

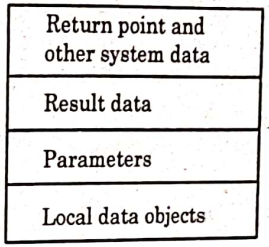


Fig. 5.4 An Activation Record for a Subprogram.

Activation records are dynamically created and destroyed during execution each time the subprogram is called and each time it terminates with a return. The resulting structure of a subprogram activation consisting of shared code and separate activation records is shown in figure 5.5.

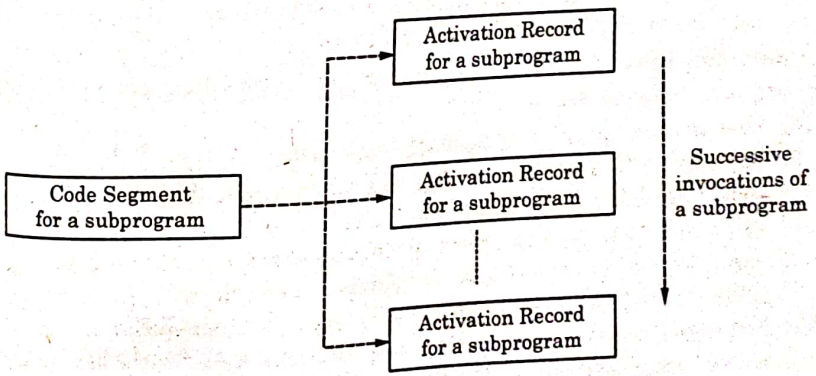


Fig. 5.5 Shared Code and Separate Activation Rerord.

The size and structure of the activation record for a subprogram can be determined during the translation. Access to the components may be made using the base-address –plus – offset calculation as described in chapter 4 for ordinary record data objects. For this reason, an activation record is in fact ordinarily represented in storage just as any other record data object.

5.7 OVERLOADED SUBPROGRAMS

Operator overloading is one of the most exciting features of object – oriented programming. It can transform complex, obscure program listings into intuitively obvious ones. An overloaded operator is one that has multiple meanings. The meaning of a particular instance of an overloaded operator is determined by the types of its operands. For example, if the operator has two floating point operands in a program, it specifies floating point multiplication. But if the same operator has two integer operands, it specifies integer multiplication.

An **overloaded subprogram** is a subprogram that has the same name as another subprogram in the same referencing environment. Every version of an overloaded subprogram must have a unique protocol (or prototype) ; *i.e.*, it must be different from others in the number, order or types of its parameters, or in its return type if it is a function. An overloaded subprogram appears to perform different activities depending on the kind of data sent to it.

Overloading is like the joke about the famous scientist who insisted that the thermos bottle was greatest invention of all time. "It's a miracle device" he said. "It keeps hot things hot, but cold things it keeps cold". This same concept is used by the overloaded functions on different data kinds of data.

As an example to illustrate the concept of overloaded subprogram. Let's define following three subprograms.

1. **starline ( )**, printing a line using 45 asterisks.
2. **repchar ( )**, using a character and a line length that are both specified when the subprogram is invoked.
3. **charline ( )**, always printing 45 characters but allowing the calling program to specify the character (asterisk) to be printed.

The above mentioned three functions perform similar activities but have different names. For programmers using these functions, that means three named to remember and three places to look them up if they are listed alphabetically in an application's **Function Reference Documentation.**

It would be far more convenient to use the same name for all three functions even though they each have different arguments. The subprogram overloading makes this possible as

1. **void repchar ( ) ;** → Always loops 45 times and prints asterisk.
2. **void repchar (char) ;** → Always loops 45 times and prints specified character.
3. **void repchar (char, ch, int n) ;** → Always loops *n* times and prints specified character

The program contains three subprograms with the same name. The first subprogram takes no arguments, the second subprogram takes one argument of type char and the third subprogram takes one argument of type **char** and another of type **int.**

The compiler uses the number of arguments, and their data types, to distinguish one function from another. The compiler, seeing several functions with the same name but different number of arguments, could decide the programmer had made a mistake (which is what it would do in C).

Instead, it very tolerantly sets up a separate subprogram for every such definition. Which one of these functions will be called depends on the number of arguments supplied in the call. Figure 5.6 shows this process.

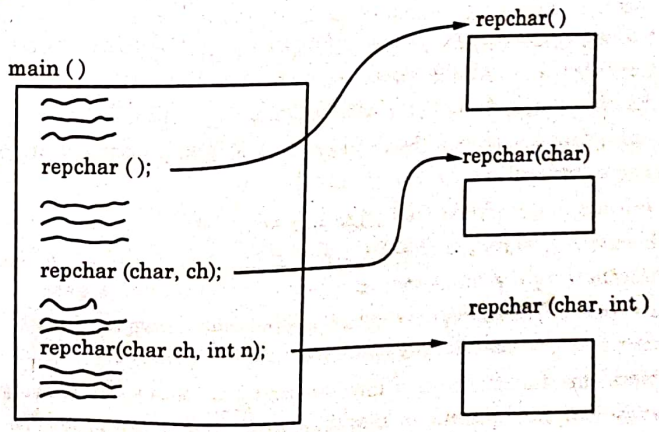


Fig. 5.6 Overloaded Subprogrms

The main advantages that an overloaded subprograms provide are :

1. Reducing the number of subprograms (functions) names to be remembered
2. Useful for handling different types of objects.

The various remarks regarding overloaded subprograms in differe programming languages are :

1. C + +, Java and Ada include predefined overloaded subprograms. F example, Ada has several versions of the output function **PUT**.
2. In Ada, the return type of an overloaded subprogram is used disambiguate calls. Therefore, two overloaded functions can have the sam parameter profile and differ only in their return types.
3. C + + and Java allow mixed-mode expressions, and the return type irrelevant to disambiguation of overloaded subprograms.
4. Users are also allowed to write multiple versions of subprogram with th same name in Ada, Java and C + +.
5. Overloaded subprograms that have default parameters can lead ambiguous subprogram calls and will cause a compilation error.

5.8 GENERIC SUBPROGRAMS

The software reuse contributes towards increase in software productivity. O way to increase the software reuse is to lesson the need to create differe subprograms that implement the same algorithm on different types of data. F example, a programmer should not need to write different sort subprograms to s different arrays that differ only in element type.

A generic or polymorphic subprogram is one with a single name but sever different definitions, distinguished by a different signature. A generic subprogra takes parameters of different types on different activations. A generic subprogra name is said to be overloaded. Overloaded subprograms provide a particular kind polymorphism is called **ad-hoc polymorphism.**

A polymorphic function has a parameterized type, also called a generic ty **Parameteric Polymorphism** is a special kind of polymorphism in which t expressions are parameterized. Parameteric polymorphism is provided by subprogram that takes a generic parameter that is used in a type expression th describes the types of the parameter of the subprogram. Data structures like stac

and queues can be defined to hold values of any type. Thus, we can define stack of integers, stack of grammer symbols, and so on. When code for such data structures is put into a library, a library designer cannot possibly anticipate all future uses of the data structure. Polymorphic types allow such a data structure to be defined once and then applied later to any desired type. Now we consider the concept of generic subprograms in different programming languages.

1. In APL, Because of the dynamic type Binding, the types of parameters can be left unspecified ; they are simply bound to the type of the corresponding actual parameters.
2. Ada provides a kind of compile time parametric polymorphism through a constructor that supports the construction of multiple versions of program units to accept parameters of different data types. The different versions of the subprogram are instantiated or constructed by the compiler upon request from the user program. Because the versions of the subprogram all have the same name, this provides the illusion that a single subprogram can process data of different types on different calls. Because program units of this sort are generic in nature, they are sometimes called **generic units.** The same mechanism can be used to allow different executions of a subprogram to call different instantiations of a generic subprogram. This is useful in providing the functionality of subprograms passed as parameters.
3. C + + provides a kind of compile-time parametric polymorphism using a constructor called templates. A type parameter T can be introduced by writing.

template < class T >

before a class or a function declaration.

Here we discuss the example of stacks. The **push** and **pop** operations on stacks do not examine the objects held in the stack, so these operations can be defined for any type. Thus, we can have stack of integer, stack of grammer symbols in a parser, and so on. In the following declaration of class stack the type parameter T denote the type of the elements held in a stack.

```
template < class T > class stack {
int top ;
int size ;
T * elements ;
public :
stack (int n) { size = n ; elements = new T [size] ; top = 0 ; }
~ stack ()  {delete elements : }
void push (T a)  { top + + ;  elements [top] = a ; }
T pop ()  { top – –; return elements [top + 1] ; }
    };
```

Program 5.1 : The declaration of a class stack.

The variable **element** will now point to the first of an array of elements of type T. In the constructor, the operator **new** is applied to T [size] to allocate an array of s, where the number of elements is given by **size**. The function **push** takes an argument of type T, and the function **pop** returns a value of type T.

For completeness, skeleton implementations of **push** and **pop** are included. The body of **push** needs to check that **top** will remain with in the array bounds before it executes **top++** to increment **top.** Similarly, the body of **push** needs to check that **top** will remain with in the array bounds before it executes **top– –** to decrement **top.**

5.9 TYPE DEFINITIONS

A type definition is mechanism for defining a new abstract data type. A type definition does not completely define an abstract data type because it does not include definition of the **operations** on data of that type. Now we see how type definitions are given in Pascal in C.

(*a*) In Pascal, a type name is given, together with a declaration that describes the structure of a class of data objects. The type name then becomes the name for that class of data objects, and when a particular data object of that structure is needed, we need give only the type name, rather than repeating the complete description of the data structure. Consider the following declaration.

```
type student = record
    id : integer ;
    age : integer ;
end
```


Now consider the declaration

```
var X, Y : student ;
```

So that the definition of the structure of a data object of type **student** is given only once rather than being repeated two times for each of X and Y.

(*b*) In C, there is a type of construct having syntax

typedef definition name

This is much like a macro substitution. Therefore, we write out student example in C as follows :

```
typedef struct student
    {
        int id ;
        int age ;
    }  s1 ;
```

Now consider the declaration

```
s1  X,  Y ;
```

which has a syntax similar to the Pascal example above. An important point to remember here is that the **struct** feature generates new types, while the **typedef** does not create a new type and only is a macro substitution of its definition.

Implementation : Another important point to remember here is that the type definition is different from type declaration. The differences between type definition and type declaration are given in table 5.2.

Type Definition	Type Declaration
1. The information contained in the definition is only used during translation. The language translator enters the information from type definition into table during translation and whenever the type name is referenced in a subsequent declaration, used the tabled information to produce the appropriate executable code for setting up and manipulating the desired data objects during execution.	1. The information contained in the declaration of a variable is used primarily during translation to determine the storage representation. For the data object and for storage management and type checking purposes.

2. The type definition allows some aspects of translation such as determining storage representations, to be done only once for a single type definition.	2. The type declaration determines storage representation, many times for different declarations.
3. Inclusion of type definitions in a language does not ordinarily change the run-time organization of the language implementation.	3. Inclusion of type declaration in a language may change the run-time organization of the language because they are used to set up the run time data objects.

Table 5.2 : Differences between Type definition and Type declaration

5.9.1 TYPE EQUIVALENCE

The type equivalence brings up two related concepts :

1. What does it mean to say that two types are **"the same"** ? and
2. What does it mean to say that two data objects of same type are **"equal"** ?

The first is a data type issue while the second is a semantic issue in determining the *r*-value of a data object.

Type equality : "A type X is same as type Y" can be true in terms of

(a) **Name equivalence :** Two data types are considered equivalent only if they have same name. Name equivalence of types is the method used in Ada, C + + and for subprogram parameters in Pascal. Name equivalence has following disadvantages :

(i) There can be no anonymous types, as every object must have a type name.

(ii) A single type definition must serve all or large parts of a program.

(b) **Structural equivalence :** Two data types are considered equivalent if they define the data objects that have same **internal components** (*i.e.* same storage representation). The structural equivalence has the disadvantages :

(i) Two variables may be structurally equivalent, even though the programmer declares then as separate types.

(ii) Determining the structural equivalence of two complex type definitions may be a costly part of translation.

(iii) Several subtle questions arise as to when two types are structurally equivalent.

Data Object equality : Once the compiler determines that two objects are the same type, are the two objects equal ? Unfortunately, the language cannot help the programmer much in this respect. Much of the design of a data type is in how it will be used. *e.g.*, if we consider that data objects X, Y, A and B are structurally equivalent types, however, the conditions upon which we may want to have X = Y or A = B will be very different.

5.9.2 TYPE DEFINITIONS WITH PARAMETERS

Where many similar type definitions are needed, a language may provide for parameterzing the type definition to be used repeatedly with different substitutions for the parameters. Array sizes have been the classical example of type parameters.

Implementation : Type definitions with parameters are used as templates during compilation, just as any other type definition, except that when the compiler translates a declaration of a variable with a parameter list following the type name, the compiler first fills in the parameter value in the type definition to get a complete type definition without parameters. Parameters in type definitions have an effect on the run-time organization of the language implementation in only a few cases *e.g.*, where a subprogram must accept any data object of the parameterized type as an argument.

5.9.2 ADVANTAGES OF TYPE DEFINITIONS

The type definitions provide the following advantages :

1. A type definition simplify the program structure.
2. A type definition is used as a **template** to construct data objects during program execution.
3. A type definition allow the separation of the definition of the **structure** of a data object from the definition of the points during execution at which data objects of that structure are to be created.
4. A type definition effectively hides the internal structure of the data objects of that type.
5. A type definition may be modified without changing the subprogram.

KEY POINTS TO REMEMBER

- The three basic mechanisms to provide the programmer with the ability to create new data types and operations on that type are **subprograms, type declarations** and **Inheritance.**
- An **abstraction** is an emphasis on the idea, qualities and properties rather than particulars *i.e.* abstraction separates the necessary details from unnecessary details.
- The two types of abstractions are **process abstraction** and **data abstraction.**
- In modern programming languages, the principle of information hiding is to make visible all that is essential for the user to know, and to hide everything else.
- **Encapsulation** is the method of information hiding or abstraction in which data or set of information, neither can be viewed by the user, nor the user can manipulate or access the hidden information.
- An **abstract data type (ADT)** is defined as a set of data values and associated operations that are precisely specified independent of any particular implementation.
- A **subprogram** is defined as a collection of statements that can be reused at several different places in a program when convenient.
- A **subprogram** has a single entry point.
- A **procedure** is a subprogram that define parameterized computations while a **function** is a subprogram that computes a value.
- A **subprogram** definition a available at compile time while the subprogram activation is available only during the program execution.
- An **activation** of a subprogram consist of
  - A fixed static part called cage segment
  - A dynamic part called an activation record.
- An **overloaded subprogram** has the same name as another subprogram in the same referencing environment but must have a unique protocol (or prototype) *i.e.* number, order or types of its parameters or return type if it is a function.

- An **overloaded subprogram** performs different activities depending upon the kind of data sent to it.
- A **generic** or **polymorphic subprogram** is one with a single name but several different definitions distinguished by a different signature.
- A **generic subprogram** takes parameters of different types, on different activations.
- A **generic subprogram** name is said to be overloaded.
- **Overloaded subprograms** provides a particular kind of polymorphism called ad-hock polymorphism.
- A **type definition** is a mechanism for defining a new abstract data type.
- Two types are same in terms of **name equivalence** and **structured equivalence.**
- A type definition is used as a template to construct data-objects during **program execution.**

EXERICISE

1. Define the term data type. Explain evolution of data types concept.
2. Explain the concept of Abstraction encapsulation and information hiding in detail.
3. What are abstract data types ? Explain in detail.
4. What are the subprograms. Explain different characteristics and advantages of subprograms.
5. What is a subprogram activation ? Explain the concept of code segment and activation records in details.
6. Differentiate between overloaded and Generic subprograms.
7. Write short notes on :
   (*a*) Data type and abstract data type
   (*b*) Procedures ands functions
   (*c*) Subprogram definition and subprogram activation
   (*d*) Type definition and Type declaration.

CHAPTER

6 SEQUENCE CONTROL

6.1 INTRODUCTION

A program regardless of the language used, specifies a set of operations applied on certain data in a certain sequence. The basic differences among the different programming languages exist in :

(a) The types of data allowed.
(b) The types of operations available, and
(c) The mechanisms for controlling the sequence in which the operations are applied to the data.

A **control structure** is a control statement and the collection of statements whose execution it controls. The control structure in a programming language provide the basic framework for combining operations and data into programs and sets of programs. The organization of data and operations into complete executable programs involves two aspects :

1. The control of the order of execution of the operations, both primitive and user defined, termed as **sequence control.**
2. The control of the transmission of data among the subprograms of a program, termed as **data control.**

6.2 IMPLICIT AND EXPLICIT SEQUENCE CONTROL

A **sequence control** is defined as the control of the order of execution of operations, both primitive and user defined. The sequence control structures direct the order of program instructions. The fact that one instruction follows another in sequence establishes the control and order of operations. A sequence – control structure may be either **implicit** or **explicit.**

Implicit Sequence Control : Implicit sequence control structures are those defined by the language. Implicit sequence control is determined by the order of the statements in the source program or by the built-in execution model. The implicit sequence control structure tends to be in effect unless modified by the programmer explicitly. The examples of an implicit sequence control includes :

(a) In many programming languages, the physical sequence of statements provide the sequence control mechanism for execution.
(b) A language defined hierarchy of operations provides the control for execution order of operations in expressions.

Explicit Sequence Control : Explicit sequence control structures are those defined by the programmer to modify the implicit sequence control defined by the language. The examples of an explicit sequence control includes :

(a) The programmer may use control statement you change the execution order *e.g.* IF statement.
(b) The programmer may use **goto** statements and statement labels to after implicit sequence control.
(c) The implicit execution order of operations in expressions can be explicitly altered by using parenthesis with in expressions.

6.3 LEVELS OF SEQUENCE CONTROL

There are three different levels at which the sequence control mechanisms may be applied :

1. **Expressions :** The sequence control mechanism at this level involves computing expressions using precedence rules, associativity rules and parenthesis. This level considers :
   (a) Sequencing with arithmetic expressions
   (b) Sequencing with non-arithmetic expressions
2. **Statements :** The sequence control mechanism at statement level controls the sequence in which the individual statements are executed with in a program. This considers
   (a) Sequential execution (or composition)
   (b) Conditional or Alteration statements
   (c) Iteration statements
3. **Subprograms :** The sequence control mechanism at subprogram level controls the sequencing between two subprograms, how one subprogram invokes another and the called subprogram returns to the first. This means that this level involves the transfer of control from one subprogram to another. This level considers
   (a) Simple Subprogram call-return
   (b) Recursive Subprograms

6.4 SEQUENCE CONTROL WITHIN EXPRESSIONS

Expressions are the fundamental means of specifying computations in a programming language. An expression consist of constants, variables, parenthesis, function calls and operators. The sequence control mechanisms with in expression considers

(*a*) Sequencing with arithmetic expressions

(*b*) Sequencing with non-arithmetic expressions

6.4.1 SEQUENCING WITH ARITHMETIC EXPRESSIONS

In programming languages, arithmetic expressions consist of operators, operands, parenthesis and function calls. The primary purpose of an arithmetic expression is to specify an arithmetic computation. The arithmetic computation (or an operation) is defined in terms of an operator and operands. *e.g.* $a + b * c$. The number of operands determines the **arity** of the operator. The operators can be **unary,** meaning that they have a single operand, or binary meaning that they have two operands. C, C + + and Java include a **ternary** operator, which has three operands. The implementation of an arithmetic computation causes following two actions.

(*a*) Fetching the operands from memory, and

(*b*) Executing the arithmetic operations on those operands

The Issue : The main issue in sequencing with arithmetic expressions is : Given a set of operations and an expression involving these operations,

(*i*) What is the sequence of performing the operations ?

(*ii*) How is the sequence defined, and how is it represented ?

6.4.1.1 BASIC SEQUENCE-CONTROL MECHANISM : FUNCTIONAL COMPOSITION

The basic sequence control mechanism in expressions is functional composition. Given an operation with its operands, the operands may be :

- Constants
- Data objects
- Other operations

For example, consider the expressions $3 * (a + 5)$. Here

Operation : multiplication (*), arity = 2 (*i.e.* binary operation)
Operand 1 : Constant (3)
Operand 2 : Operation addition
Operand 1 : data object (*a*)
Operand 2 : Constant (5)

The functional composition imposes a **tree structure** on the expressions, where we have :

(*i*) One main operation represented by root node of tree, decomposable into an operator and operands.

(*ii*) Intermediate level operations represented by nodes between the root and the leaves.

(*iii*) Data references (or constants) represented by the leaf nodes.

The tree representation of the expression $3 * (a + 5)$ is given as shown in figure 6.1.

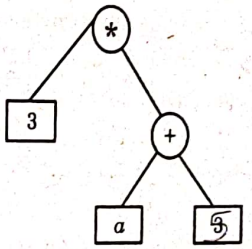


Fig. 6.1 Tree Representation of an Expression 3*(a+5)

The tree representation of an expression clearly show that the results of data references or operations at the lower levels in the tree serves as operands at higher level in the tree, and thus these data references and operations must be evaluated first. For example figure 6.1 shows that the expression $(a + 5)$ act as an operand for next higher level operation $(3 * (a + 5))$, therefore it must be evaluated first.

In a parenthesized expression the main operation is clearly indicated. However we may have expressions with parenthesis. Therefore, it is crucial for a programmer to understand both the syntax and semantics of expressions.

6.4.1.2 SYNTAX FOR EXPRESSIONS

In arithmetic expressions the standard precedence and associativity of operations are applied to obtain the tree structure of the expression. The linear representation of the expression tree provides following notations for writing the syntax of expressions :

(*a*) The Prefix notation

(*b*) The Post fix notation

(*c*) The Infix notation

These notations are described as given below :

(a) **The prefix notation :** This notation is also known as **Polish prefix** notation. In **prefix notation** an operator is written first, followed by the operands in order from left to right. If an operand is itself an operation with operands, then the same rules apply. For example the prefix notation of an expression represented by figure 6.1 is $* 3 + a5$. A variant of the prefix notation is Cambridge Polish (used in LISP) in which a parenthesis surround an operator and its arguments. *e.g* the Cambridge Polish notation of expression represented by figure 6.1 is $(* 3 (+ a 5))$. The prefix notation is sometimes also called **parenthesis free** because the operands of each operator can be found unambiguously without the need for parenthesis. In general an expression in prefix notation is written as follows.

- The prefix notation for a constant or a variable is the constant or variable itself.
- The application of an operator **op** to sub expressions E_1 and E_2 is written in prefix notation as **op** E_1 E_2.

The main advantage of prefix notation is that it is easy to decode during a left-to-right scan of an expression.

(b) **Postfix Notation :** The **postfix notation** is also known as **Reverse polish** notation. In a **postfix notation** the operator follows the list of operands. An expression in postfix notation is written as follows :

- The postfix notation for a constant or a variable is the constant or variable itself.
- The application of an operator **op** to sub expressions E_1 and E_2 is written in postfix notation as E_1 E_2 **op.**

For example, the postfix notation of an expression represented by the tree of figure 6.1 is $3a\ 5\ +\ *$. The main advantage of postfix expressions is that they can be mechanically evaluated with the help of a stack data structure. Like, prefix notation, the postfix notations are also sometimes called **parenthesis free** because, the operands of each operator can be found unambiguously without the need for parenthesis.

(c) **Infix Notation :** In an **infix notation**, the operator appears between their operands. An expression in infix notation is written as follows :

- The infix notation for a constant or a variable is the constant or variable itself.

- The application of an operator **op** to sub expressions E_1 and E_2 is written infix notation as E_1 **op** E_2.

For example, the infix notation of an expression represented by the tree of figure is $3 * (a + 5)$. The main advantage of an infix notation is that it is familiar and once easy to read. Moreover, an infix notation comes with the rules for precedence and associativity which helps in answering the question like whether $a + b * c$ is to decoded as $a + (b * c)$ or $(a + b) * c$.

6.1.3 SEMENTICS FOR EXPRESSIONS

The three notations (Prefix, postfix and infix) differ in how one can compute the value for each expression. In this section we give algorithms for evaluating (*i.e.* computing the semantic) for each expression format.

(a) **Prefix Evaluation :** With prefix notation, one needs to know the number of arguments for each operation. Given a prefix expression P consisting of operators and operands, the prefix expression can be evaluated with an execution stack as :

(*i*) If the next item in P is operator, push it on the top of stack and set the argument count to the number of operands required for the operator.

(*ii*) If next item in P is an operand, push it on the top of stack.

(*iii*) If there are required number of operands in the top of stack then apply the top operator to these operands and replace the operator and its operands by the result of applying that operation on its operands.

For example the sequence of steps for evaluating the prefix expression $* 3 + a5$ is shown in figure 6.2.

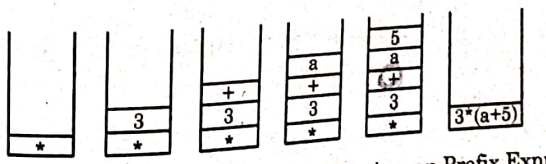


Fig. 6.2 A Sequence of Stack Actions for Evaluating an Prefix Expression *3 + a5

The main problem with prefix evaluations is that after pushing an operand on the stack, we still have to check if we have enough operands to satisfy the current top operator.

(b) **Postfix Evaluation :** The postfix evaluation overcomes the limitation of the prefix evaluation described above. Given a postfix expression P

consisting of operators and operands, the postfix expression can evaluated using a stack as given below :

(*i*) If the next item of P is an operand, push it on the top of stack.

(*ii*) If the next item of P is an *n*-ary operator, then pop items from the of stack, apply the operator and replace the *n* items and operator by result of applying that operation on *n* operands.

For example, the sequence of steps for evaluating the postfix expres 3*a* 5 + * is shown in figure 6.3.

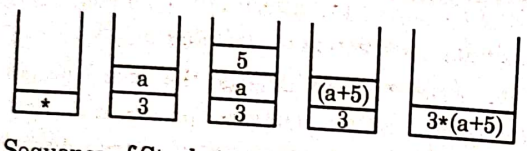


Fig. 6.3 A Sequence of Stack Action for Evaluating a PostfixExpression 3a5+*

The postfix evaluation strategy is straight forward and easy to implement. I fact, it is the basis for generating code for expressions in many translators.

(c) Infix Evaluation : In infix notation, operators appear between their operands. The main advantage of infix notation is that it is familiar and hence easy to read. But its use in a programming language leads to a number of problem as :

(*i*) Infix notation is suitable only for binary operators therefore a language cannot use only infix notation but must combine infix and prefix (or postfix) notation.

(*ii*) The infix notation is inherently ambiguous unless parenthesis are used.

For example consider the decoding of an expression like $a + b * c$. This expression may be decoded in following two ways.

- Sum of a and $b * c$.
- Product of $a + b$ and c.

However parenthesis may be used to disambiguate the expressions like above indicating either $(a + b) * c$ or $a + (b * c)$, but in complex expressions the resulting deep nests of parenthesis become confusing.

The above ambiguities can be resolved by introducing the concepts of precedence and associativity of operators. There are implicit control rules that make most uses of parenthesis unnecessary. These rules are described as :

Operator precedence rules (Hierarchy of operations) : The operator precedence rules for expression evaluation define the order in which the operators of different precedence levels are evaluated. The operator precedence rules for expressions are based on the hierarchy of operator priorities, as seen by the language designer.

The operator precedence rules of the common imperative languages are nearly all the same, because they are all based on those of mathematics. In these languages, exponential (denoted by ** or ↑) has the highest precedence (when it is provided by the language), followed by multiplication and division on the same level, followed by binary addition and subtraction on the same level.

In all of the common imperative languages, the unary minus operator can appear in an expression either at the beginning or anywhere inside the expression, as long as it is parenthesized to prevent it from being adjacent to another operator. For example,

$$A + (-B) * C$$

is legal, but

$$A + -B * C$$

usually is not

The precedences of the arithmetic operators of a few common programming languages are as shown in table 6.1.

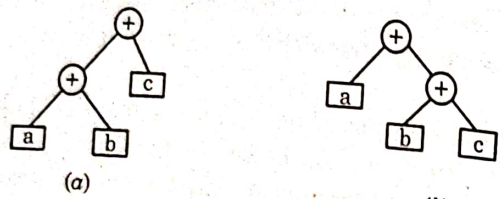
	FORTRAN	PASCAL	C	ADA
Highest	* *	*, /, div, mod	postfix + +, – –	* *, abs
	*, /	all +, –	prefix + +, – –	*, /, mod
	all +, –		unary +, – –	unary +, –
			*, /, %	binary +, –
Lowest			binary +, –	

Table 6.1 : Operator precedences for a few common programming languages

For example the expression $a + b * c$ is correctly interpreted as $(a + (b * c))$ according to the rules of operator precedence in C.

Associativity of operators : The precedence accounts for only some of the rules for the order of operator evaluations. For example, consider the expression $a + b + c$. As the operators used in this expression belongs to a single level, the precedence rules say nothing about the order of evaluation of the operators in this

expression. The expression $a + b + c$ may be interpreted as $(a + b) + c$ or $a + (b$, as shown in figure 6.4.



(a) (b)

Fig. 6.4 Two Different Interpreteting of Expression a+b+c

When an expression contains two adjacent occurrences of operators with the same level of precedence, the question of which operator is evaluated first is answered by the **associativity** rules of the language. An operator can either have

(*i*) Left associativity, meaning that the leftmost occurrence is evaluated first or

(*ii*) Right associativity, meaning that the right most occurrence is evaluated first.

Associativity in common imperative languages is left to right, except that the exponentiation operator (when provided) associates right to left.

When unary operators appear at positions other than at the left end of expressions, they must be parenthesized in that language, so in those situations those operators are forced to have the highest precedence. The associativity rules for a few of the most common imperative languages are as given in table 6.2.

Language	Association Rule
FORTRAN	Left : *, /, +, –
	Right : * *
PASCAL	Left : all
	Right : None
C	Left : postfix + +, postfix – –, *, /, %, binary +, binary –
	Right : prefix + +, prefix – –, unary +, unary –
C + +	Left : *, /, %, binary +, binary –
	Right : + +, – –, unary –, unary +
Ada	Left : all except * *
	Non associative : * *

Table 6.2 : Associativity rules for a few common programming languages

For example, the expression $a + b + c$ is correctly interpreted as $((a + b) + c)$ according to the rules of the operator associativity in C.

6.4.1.4 EXECUTION – TIME REPRESENTATION

Because of the problem evaluated with the decoding of expressions in the infix form (as discussed in section 6.4.1.3), it is common to translate it into an executable form that may be easily decoded during execution. The following are the most important alternatives.

(*i*) Machine code sequences, performing the two stages of translation in a single step.

(*ii*) Tree structures, which can represent an expression using a software interpreter. The execution may then be accomplished by a single tree traversal.

(*iii*) Prefix or postfix form, which can execute an expression using a stack organization.

6.4.1.5 EVALUATION OF TREE REPRESENTATIONS OF EXPRESSIONS

The most important problem of order of evaluation is one in which the tree is translated into an executable sequence of primitive operations. The different problems of order of evaluation that arise in determining exactly the code to generate are as follows :

1. **Uniform Evaluation rules :** There are following two uniform evaluation rules.

(*a*) **Eager Evaluation :** This rule states that for each operation node, in the expression tree, first evaluate each of its operands and then apply the operation to the evaluates operands. The rule is called **eager evaluation rule** because we always evaluate operands first. The main limitation of an eager evaluation rule is in case of expressions containing conditionals e.g. the C expression Z + (Y = 0 ? X : X/Y) has an embedded if that computes X/Y if Y is not 0. If we assume eager evaluation rule and evaluate the operands of conditional operator, we may produce the effect of dividing X/Y even if Y is zero.

(*b*) **Lazy Evaluation rule :** The problem with conditionals suggest an alternative uniform evaluation rule, called a **lazy evaluation rule** which states that : Never evaluate operands before applying the operation ; instead always pass the operands unevaluated and let the operation decide if evaluation is needed. This evaluation rule works in all cases and thus theoretically would serve.

6.4.1.6 PROBLEMS

Some of the problems encountered are discussed below :

1. **Side effects :** A side effect of a function (or operation), called a function side effect, occurs when the function changes either one of its parameters a global variable. (A global variable is declared outside the function but accessible in the function).

Consider the expression

A + FUN (A)

If FUN does not have the side effect of changing A, then the order of evaluation of the two operands, A and FUN (A), has no effect on the value of the expression. However, if FUN changes A, there is an effect. For example, consider the following situation : FUN returns the value of its argument divided by 2 and changes its parameter to have the value 20. Suppose we have the following

A : = 10 ;
B : = A + FUN (A)

Then, if the value of A is fetched first (in the expression evaluation process) its value is 10 and the value of the expression is 15. But if the second operand is evaluated first, then the value of the first operand is 20 and the value of the expression is 25.

There are two solutions to the problem of operand evaluation order.

(*a*) The language designer could disallow function evaluation from affecting the value of expression, by simply disallowing functional side effects.

(*b*) To state in the language definition that operands in expressions are to be evaluated in a particular order and demand that implementers guarantee that order.

There is no dominant position on side effects in expression ; either approach has its adherents.

2. **Error Conditions :** A number of errors can occur in expression evaluation. If the language require type checking, then operand type errors cannot occur. The error conditions are a special kind of **side effects**. However, the side effects are generally restricted to programmer – defined functions while the error conditions may arise in many primitive operations. some of error conditions that may occur are

(*i*) Errors that can occur because of coercions of operands in expressions.

(*ii*) Floating – point overflow and underflow.

(*iii*) Divide by zero error.

The solution to there difficulties are adhoc and varies from language to language and implementation to implementation.

3. **Short – Circuit Boolean Expressions :** A short-circuit evaluation of an expression is one in which the result is determined without evaluating all of the operands and/or operators, for example, the value of the arithmetic expression.

(16 * A) * (B/16 – 1)

is independent of the value of (B/16 – 1) if A is 0 then $0 * x = 0$ for any x. Therefore there is no need to evaluate (B/16 – 1) or perform the second multiplication. However, in arithmetic expressions this short is not easily detected during execution, so it is never taken.

In programming it is often used for the Boolean operations and (& in C) and or in C) to combine relational expressions such as C statement.

e.g. (A > = 0) && (B < 10)

is independent of the second relational expression. If the first condition (A > = 0) false then the overall express is false (an && operation requires both the condition be true) and there is no need to evaluate second condition. Consider another, ample

(A > = 0) || (B/A < 10)

Here if the first condition (A > 0) is true, the overall condition is true (an || eration is true if one of the conditions is true) and there is no need to evaluate cond condition (B/A < 10).

The short circuit evaluation of expressions exposes the problem of allowing side ects in expressions. Suppose that short-circuit evaluation is used on an expression d part of the expression that contains a side effect is not evaluated ; then the side fect will only occur in complete evaluations of the whole expression. If the program rrectness depends on the side effect, short-circuit evaluation can result in a serious ror.

In C, C + + and Java, the usual AND and OR operators, && and || respectively e short-circuit. However, these languages also have bitwise AND and OR erators, & and 1, respectively, that can be used on Boolean valued operands and e not short circuit.

The inclusion of both short-circuit and ordinary operators is clearly the best sign because it provides the programmer the flexibility of choosing short-circuit valuation for any on all Boolean expressions.

6.4.2 SEQUENCING WITH NON–ARITHMETIC EXPRESSIONS

In the previous subsection we have discussed about arithmetic expressi... However many programming languages has the form of expressions made up variables, constants, pair or tuples and list constructors. Value constructors for d... types, can also appear in expressions. *e.g.* the languages like ML and Prolog incl... a form of expression evaluation, designed for processing character data.

6.4.2.1 PATTERN MATCHING

Pattern matching is a fundamental character string operation in m... programming languages like SNOBOL4, PROLOG, ML and PERL etc. It is oft... provided by a library function rather than as an operation in the language. T... language SNOBOL4 has an elaborate pattern-matching operation built into t... language and is probably the ultimate string manipulation language. The SNOBO... use string replacement for its pattern matching.

In order to illustrate the concept of pattern matching in SNOBOL4, consider t... following grammer that recognizes odd-length palindromes over $\sum = \{a, b\}$.

$$S \rightarrow a\,S\,a \mid b\,S\,b \mid 0 \mid 1$$

Now consider the $w = b\,b\,a\,b\,b$ recognition of a string

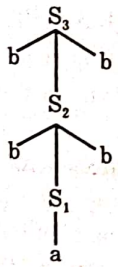


Fig. 6.5 SNOBOL 4 Pattern Matching.

The process of pattern matching takes place as

(*a*) S_1 matches the centre a.

(*b*) S_2 matches $b\,S_1\,b$

(*c*) S_3 matches $b\,S_2\,b$

From this three "assignments" of S_1 to a, S_2 to $b\,S_1\,b$ and S_3 to $b\,S_2\,b$, ... construct the parse tree of the string as shown in figure 6.5.

As another example of string patterns in SNOBOL4 expressions, consider the following :

LETTER = '*abcdefghijklmnopqrstuvwxyz*'

WORDPAT = BREAK (LETTER) SPAN (LETTER) . WORD

LETTER is a variable with the value of string of all lower case letters. WORDPAT is a pattern that describes words as follows : first skip until a letter is found, then span those letters until a non letter is found. This pattern also includes a "." operator, which specifies that the string that matches the pattern is to be assigned to the variable WORD. This pattern can be used in the statement

TEXT WORDPAT

which attempts to find a string of letters in the string value of the variable, TEXT.

While SNOBOL4 uses string replacement for its pattern – matching operation, PROLOG uses the concept of a relation as a set of *n*-types as its matching mechanism. By specifying known instances of these relations (called **facts**), other instances can be derived. Infact, Matching a PROLOG query with a rule in the database is a form of term rewrite (with substitution). The concept of term rewrite is : "Given string $a_1\,a_2 \ldots a_n$ and rewrite rule $\alpha \Rightarrow \beta$, if $\alpha = a$; we say $a_1 \ldots a_{i-1}\,\beta \ldots a_n$ is a term rewrite of $a_1\,a_2 \ldots a_n$. For example given the grammer

$$S \rightarrow aA/b$$

$$A \rightarrow a\,S$$

We can generate the string $a\,a\,b$ with the following derivation

$S \Rightarrow a\,A$ using rule $S \rightarrow a\,A$

$a\,A \Rightarrow a\,a\,S$ using rule $A \rightarrow a\,S$

$a\,a\,S \Rightarrow a\,a\,b$ using rule $S \rightarrow b$

6.4.2.2 UNIFICATION

A prolog database consists of facts and rules where

(*a*) **Fact** : The facts are relationships that are stated during a **consult** operation, which adds new facts and rules to database. They are *n*-tuples $f\,(a_1\,,\,a_2\,,\,\ldots\,a_n)$ and state the relationship among the *n* arguments according to relationship *f*. *e.g* the fact

father (A, B);

simply means that A is father of B.

(b) Rules : The rules are implications that are stated during a consult operation. The rule statement in PROLOG corresponds to a **headed Horn-Clause** which can be related to a known theorem in mathematics from which a conclusion can be drawn if the set of given conditions is satisfied. The right side is the antecedent, or **if** part, and the left side is the consequent, or **then** part. If the antecedent of a PROLOG statement is true, then the consequent of the statement must also be true. The general form is given as

Consequence _1 : – antacedent_expression

e.g **ancestor** (mary, bill) :– **mother** (mary, bill)

States that if mary is the mother of bill, then mary is an ancestor of bill. Headed Horn clauses are called **rules** because they state rules of implication between propositions (declarative statements that are either true or false).

(c) Goal (or query) : A goal or a query is an expression containing one or more variables *e.g* in father of (X, Mary).

(d) Resolution : Resolution is an inference rule that allows inferred propositions to be computed from given propositions, thus providing a method with potential application to automatic theorem proving.

Prolog uses **unification,** or the **substitution** of variables in relations, to pattern match in order to determine if the query has a valid substitution consistent with the rules and facts in the database. Unification allows variables to be used as place holders for data to be filled in later. For example consider that we have the following query to solve

$$f(X, b) = f(a, Y)$$
$$X = a$$
$$Y = b$$

An **instance** of a term T is obtained by substituting subterms for one or more variables of T. The same substitution must be substituted for all occurences of a variable. Thus $f(a, b)$ is an instance of $f(X, b)$ because it is obtained by substituting subterm a for variable X in $f(X, b)$. Similarly $f(a, b)$ is an instance of $f(a, Y)$ because it is obtained by substituting subterm b for variable Y in $f(a, Y)$. Therefore we have the fact that $f(a, b)$ is a solution to both parts of our-query.

In general the concept of unification used for deduction (or inferencing) in PROLOG is defined as :

"The two terms T_1 and T_2 unify if they have a common instant U. If a variable occurs in both T_1 and T_2, then the same subterm must be substituted for all occurrences of the variable in both T_1 and T_2". For example the two terms $f(X, b)$ and $f(a, Y)$ unify because they have common instance $f(a, b)$. Unification occurs implicitly when a rule is applied.

The process of **unification** can be thought of as an extension to the common property of **substitution**. The differences between unification and substitution are given in the tabular form.

Substitution	Unification
1. Substitution means that a string is literally substituted and is the general principle behind parameter passing and macro expansion.	1. Unification can be thought of as an extension to the common property of substitution and is the process of determines the useful (valid) values for variables.
2. Substitution is the result of applying new values to macro template arguments.	2. Unification is the result of simultaneous substitutions to multiple macro templates in order to show that all are equivalent under some set of simultaneous substitution.
3. When we apply substitution, we have soma pattern definition which may represent a subprogram signature or a macro definition, and an instance of the pattern, which may represent the invocation of the subprogram or a macro expansion.	3. When we apply unification, we usually have two separate pattern definitions and an instance of a pattern.

Table 6.3 : Differences between substitution and unification

6.4.2.3 BACKTRACKING

Backtracking is a general programming technique available in any language that creates tree structures. We can build backtracking algorithms in all of the languages. But in languages like LISP, trees are natural consequences of the built-in list data type, backtracking is relatively easy to implement.

In Prolog, when a goal with multiple subgoals is being processed and the system fails to show the truth of one of the subgoals, the system abandons the subgoal it could not prove. Instead, the system reconsiders the previous subgoal, if there is one,

and attempts to find an alternative solution to it. This backing up in the goal to the reconsideration of a previously proven subgoal is called **backtracking**. A new solution is found by beginning the search where the previous search for that subgoal stopped. Multiple solutions to a subgoal result from different instantiations of its variables. Backtracking can require a great deal of time and space because it may have to find all possible proofs to every subgoal. These subgoal proofs may not be organized to minimize the time required to find the one that will result in the final complete proof, which exacerbates the problem.

To understand the concept of backtracking assume that there is a set of facts and rules in the database and prolog has been presented with the following compound goal :

male (X), **Parentof** (X, John)

This goal asks whether there is an instantiation of X such that X is a male and X is a parent of John. The following steps are performed.

1. Find the fact in the database with **male** as its function and instantiate X to any parameter say Mike.
2. Attempt to prove that parent (Mike, John) is true,
3. If it fails, it backtracking to the first subgoal, male (X) and attempt to the subgoal with another alternative instantiation of X in male (X).
4. The resolution process may have to find every **John** before it finds the one that is parent of John. It definitely must find all **males** to prove that the goal cannot be satisfied.
5. The example goal can be processed more efficiently in we reverse the order of two subgoals. Then, only after resolution had found a **parent** of John would it try to match that person with the **mail** subgoal. This is more efficient if **John** has fever parents than there are **males** in the database which seems like a fair assumption.

6.5 SEQUENCE CONTROL WITHIN STATEMENTS

The basic units of imperative programming are actions, which can change the values of variables. The constructs called statements specify actions and the flow of control around actions. The statements apply the operations to data objects. The basic statements in a language generally includes :

(*a*) Assignment statements
(*b*) Input and output statements
(*c*) Subprogram calls

The basic introduction to these statements is provided here.

(*a*) **The Assignment Statement :** The assignment statement has already been discussed in chapter 2. The primary purpose of an assignment statement is to assign to the *l*-value of a data object, the *r*-value of some expression. The assignment statement results in changing the values of variables. For example, the following assignment changes the value of variable *x* :

The assignment symbol : = appears between the variable *x* and the expression 2 + 3. This assignment computes the value 5 of the expression 2 + 3 and associates it with *x*; the old values of *x* is forgotten.

The syntax of an explicit assignment statement varies widely in different programming languages. The table 6.4 shows the syntax of assignment statements used in different programming languages.

SYNTAX	CORRESPONDING PROGRAMMING LANGUAGES
A : = B	Pascal, Ada
A = B	C, FORTRAN, PL/I, Prolog, ML, SNOBOL4
MOVE B To A	COBOL
A ← B	APL
(SETQ A B)	LISP

Table 6.4 : Different syntax of assignment statements

In many programming languages a single assignment operator is used. However C, uses several assignment operators as shown in table 6.5.

SYNTAX	SEMANTICS
A = B	Assign toil-value of A, the *r*-value of B, return *r*-value
A + = B	A = A + B (*i.e.* increment A by B) and return new value
A – = B	A = A – B (*i.e.* decrement A by B) and return new value
+ + A	A = A + 1 (*i.e.* increment A by 1), return new value
– – A	A = A – 1 (*i.e.* decrement A by 1), return new value
A + +	Return the *r*-value of A, increment A by 1 (*i.e.* A = A + 1)
A – –	Return the *r*-value of A, decrement A by 1 (*i.e.* A = A – 1)

Table 6.5 : Syntax and semantics of different C assignment operators

(b) **Input-output Statements :** Many programming languages includes statements for reading data from the user at terminal, from filer etc, and unting (or displaying) data at terminals. For example the statement read (file, data) reads, data from a specified file. Similarly in C, a call of the **printf** function causes the assignment to the **"buffer variable"** of a file.

(c) **Subprogram Calls :** These statements are discussed in section 6.6.

6.5.1 FORMS OF STATEMENT – LEVEL SEQUENCE CONTROL

The three main forms of statement – level sequence control are usually distinguished :

1. **Composition :** The statements are executed in the order they appear on the page.
2. **Alternation :** Two sequences form alternatives so one sequence or the other sequence is executed but not both. A **conditional statement** is one that expresses alternation of two or more statements or optional execution of a single statement ; where statement means either a single basic statement, a compound statement, or another control statement.
3. **Iteration :** A sequence of statements that are executed repeatedly. **Iteration** provides the basic mechanism for repeated calculations in most programs.

6.5.2 EXPLICIT SEQUENCE CONTROL BETWEEN STATEMENTS

The explicit sequence control between statements can be provided by using :

(a) GO TO statement

(b) Break statement or

(c) Continue statement

These are described briefly as

(a) **GOTO statement :** A GOTO statement in a program interrupts the normal flow of control from one statement to the next in sequence and transfers the execution control to a specified place in the program. Two forms of **goto** statement are generally present in many programming languages.

(i) **Unconditional goto :** The unconditional goto transfers the control to the labeled statement *e.g.* the statement go to X transfers the control the statement labelled X.

(ii) **Conditional goto :** The condition **goto** statement transfers the control to the labeled statement only if the specified condition holds *e.g* the conditional go to statement if Y, goto X, transfers the control the statement labelled X is Y is true.

(b) **Break Statement :** Some programming languages like C use the **break** statement which causes control to move forward in the program to an explicit point at the end of a given control structure. The **break** statement in C causes control to exit the immediately enclosing **while, for** or **switch** statement.

(c) **Continue Statement :** Some programming language, like C use the **continue** statement which causes control to move forward to the end of the current loop body in a **while** or **for** statement.

6.5.3 STRUCTURED PROGRAM DESIGN

Many programming languages uses **goto** statements to provide explicit sequence. The main advantages of **goto** statements are :

1. Simple and easy to use.
2. Familiar to programmers trained in assembly language or adder languages.
3. Immediate transfer of control.
4. Direct hardware support for efficient execution.
5. Completely general purpose as a building block for simulating (representing) any of the other control forms.

However, the disadvantages of using goto statements are

1. **Ureadable programs :** Without restriction on use, imposed either by language design or programming standards, **goto** statements can make programs virtually unreadable, and all a result, highly unreliable and difficult to maintain.
2. **Lack of hierarchical program structure :** In a program design, the hierarchical organization is essential to allow the programmer to comprenend how all the parts of the program fit together. But in a program containing **gotos,** the hierarchical structure is largly obscured. Rather the program appears to have a very **"flat"** structure.
3. **Multiple-input-output control structures :** The concept of the one-in-one-out (*i.e.* single entry, single exit) makes for a more understandable design. However, a program with gotos generally may have multiple entries

and multiple exists in a program which makes it difficult to understand for a programmer.

4. **Order of statements in the program text need not correspond to the order of execution :** Using gotos, it is easy to write programs in which control jumps between different sequences of statements in irregular patterns. Therefore, there is very little connection in the order of statements and the execution sequence of statements.
5. **Groups of statements may serve multiple purposes :** Using gotos one can combine different group of statements so that the identical statements are written only once, and the control is transferred to this common set during the execution of each group. The multipurpose nature of statements tends to make the program difficult to modify.

After discussing the disadvantages of the gotos, we discuss the program design principle known as structured programming.

The principle of structured programming states that the structure of the program text should help us to understand what the program does. The structured programming design emphasize on :

1. Single entry, single exist (one-in, one-out) control structure.
2. Making the programs readable that can make them easy to modify and tune for efficiency.
3. Hierarchical design of program structures using only composition, alteration and iteration structures.
4. Program text in which textual sequence of statement corresponds to the execution sequence of statements.
5. Use of single-purpose groups of statements.
6. To write the programs easy to understand, debug, verify to be correct, and later modify and reverify.
7. Representing the hierarchical design by structured sequence control statements.

6.5.4 STRUCTURED SEQUENCE CONTROL

One of the most important aspects of a structured program design is to use **single-entry, single exit (one-in, one-out)** structure. The structured program design make use of following three control forms to impose hierarchical structured program design.

1. Compound statements
2. Conditional statements
3. Iteration statements

These statements are described in brief in following subsections.

6.5.4.1 COMPOUND STATEMENTS

Specification : The compound statements allow a collection of statements to be abstracted to a single statement. The collection of statements is generally enclosed by the keywords **begin** and end (like in Pascal) or { and } (as in C). The typical syntax is

```
begin
        statement 1 ;
        statement 2 ;
        -----
end ;
```

with in the compound statement, the textual sequence of statements corresponds to the execution sequence of statements. Thus the compound statement represents composition of statements as shown in the figure 6.6.

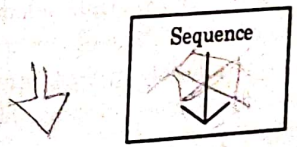


Fig. 6.6 Flow of Control in Compound Statement

Implementation : A compound statement is implemented in computer by placing the blocks of executable code representing each constituent statement in sequence in memory. As described earlier the physical sequence of statements in memory determines the execution order of the statements.

6.5.4.2 CONDITIONAL STATEMENTS

Specification : A conditional statement is used to represent alteration of two or more statements. The conditional statements are used to alter the flow of control when a choice needs to be made between two or more actions. Often the choice is based on the state of some variables in the program. The conditional statements can be statements expressed using :

(a) Two-way alteration (selection) statements

(b) Multiple selection constructs

(a) Two-way alteration statements : The two-way alteration statement selects one of two alternative substatements for execution. This control structure is commonly specified using the keywords **if** and **else**. The typical syntax of an **if** statement is

if < condition > **then** < statement 1 > **else** (statement 2 >.

If expression is true, then control flow through statement 1 ; otherwise, control flows through statements as shown in figure 6.7.

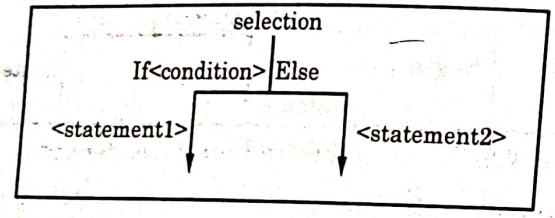


Fig. 6.7 Flow of Control in two-way Alternation Statements

A variant of the above mentioned syntax is

if < condition > then < statement >

with no **else** part. Here statement is executed only if the condition is true.

(b) Multiple Selection Constructs : The multiple selection constructs allows the selection of one of any number of statements or statement groups. It is, therefore, a generalization of a selector. In fact, single-way and two-way selectors can be built with a multiple selector. The multiple selection constructs can be expressed using

(i) Nested **if** statements

(ii) **Case** statements

(i) Nested if statements : A choice among many alternatives may be expressed be nesting additional **if** statements with in the alternative statements of a single **if**, or by a multibranch if. The typical syntax of a nested **if** statement is

```
if < condition 1 > then < statement 1 >
    else if < condition 2 > then < statement 2 >
    .....
    else if < condition n > then < statement n >
    else < statement n + 1 > end if
```


The concept is shown in figure 6.8.

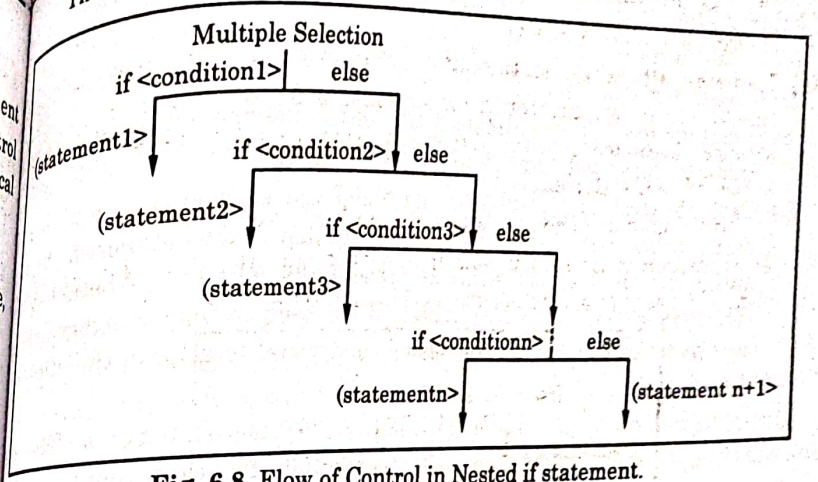


Fig. 6.8. Flow of Control in Nested if statement.

(ii) Case Statements : A **case** statement uses the value of an expression to select one of several substatements for execution. Thus, a case statement is made up of an expression and a sequence of cases, where each case consists of a constant and a substatement. The typical syntax of a **case** statement in Pascal is

```
case < expression > of
< constant 1 > : < statement 1 > ;
< constant 2 > : < statement 2 > ;
      ------
< constant n > : < statement n >
end
```

Here, the execution begins with the evaluation of **<expression>**, If its value equals that of one of the constants, say < constant *i* >, then control flows to the corresponding < statement *i* >. After execution of the selected substatement, the control leaves the case statement. The concept is shown in figure 6.9.

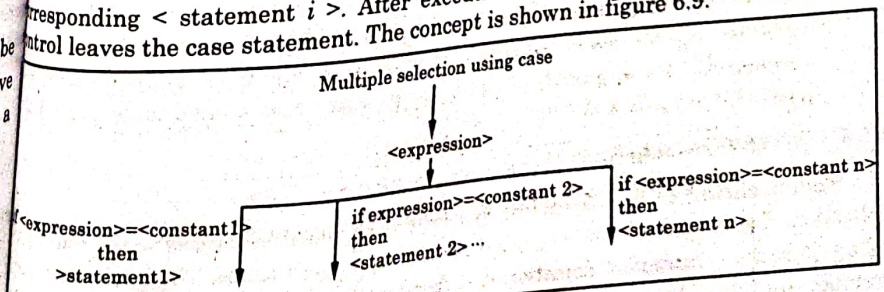


Fig. 6.9 Flow of Control in a Case Statement.

Implementation : The following two implementations are considered :

(*a*) *if* statements

(*b*) **Case** statements

(*a*) *if* statements : The **If** statements are readily implemented using the usu hardware supported branch and jump instructions.

(*b*) Case Statements : The implementation of case statements can affect the usage. Some implementations recommend that a case statement be use only when the case constants are essentially adjacent. Where the case constants are not adjacent, conditional are used instead. Furthermore, a else part can be added to the nested conditional to achieve the effect of default case.

Other implementations, encourage the use of case statements. The code generated by good compilers depends on the distribution of case constants :

(*i*) A small number of cases (say less than seven) is implemented using conditionals. If the first condition is true, then do the first substatement ; if the second condition is true, then do the second substatement, and so on.

(*ii*) For a larger number of cases, the range in which the constants appear is checked to see if an array constituting a "jump table" can be used. A jump **table** is a vector, stored sequentially in memory, each of whose component is an inconditional jump instruction. The entry *i* in the jump table is a machine instruction that sends control to the code for case *i*. The value of the expression becomes an index into the jump table, so selection of each case occurs efficiently by indexing into the table and then jumping to the code for the case. If the smallest constant is **min** and the largest is **max**, then the jump table has **max-min + 1** entries. Of course, only the entries for the case constants that actually appear are used. The compiler in question uses a jump table if atleast half the entries will be used.

(*iii*) Finally, if the number of cases is large enough, and if too many entries in a jump table would remain unused, the compiler uses a hash table to find the code for the selected substatement.

6.5.4.3 ITERATION STATEMENTS

Specification : An **iteration statement** is one that causes a statement or collection of statements to be executed zero, one, or more times. The basic structure of an iteration statement consists of a head and a body. The head controls the number of times that the body will be executed, while the body is usually a

compound statement that provides the action of the statement. The variants of an iteration statement are :

(*a*) Definite iteration

(*b*) Indefinite iteration

These are described as given below :

(*a*) Definite iteration : A definite iteration is executed a predetermined number of times. The different variants of definite iteration are

(*i*) Simple repetition

(*ii*) Repetition while condition holds

(*iii*) Repetition while incrementing a counter

(*i*) Simple iteration : The simplest type of iteration statement head specifies that the body is to be executed some fixed number of times. For example, the COBOL PERFORM statement is :

perform body K **times**

which causes K to be evaluated and then the body of the statement to be executed that may times as shown in figure 6.10.

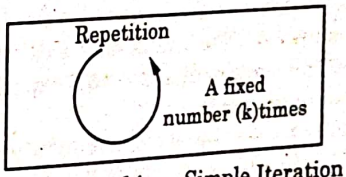


Fig. 6.10 Flow of Control in a Simple Iteration Statement

(*ii*) Repetition while condition holds : This iteration construct may be expressed using **while ... do**, and **repeat ... until** constructs

The syntax of a while ...do statement is

while < expression > **do** < statement >

This means evaluating the < expression > and if true, executes < statements >, then repeat process.

The syntax of repeat ...until is given as

repeat < statement > **until** < expression >

This means execute < statement > and then evaluate < expression >, until the < expression > is true. The concept is shown in figure 6.11.

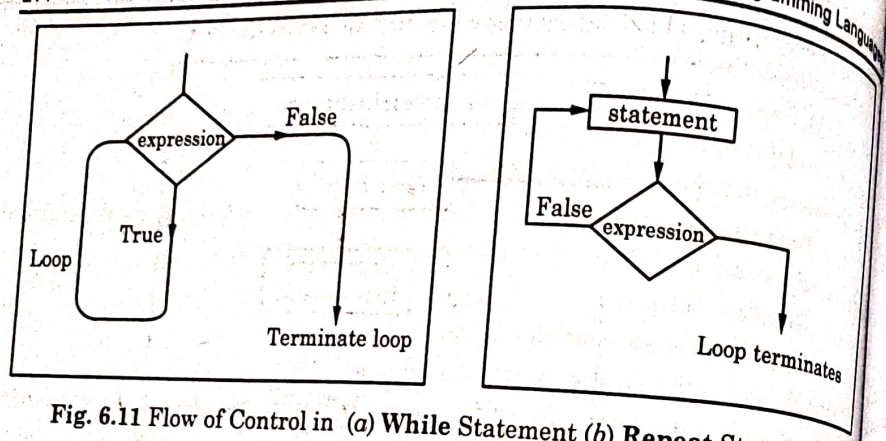


Fig. 6.11 Flow of Control in (*a*) **While** Statement (*b*) **Repeat** Statement

(iii) **Repetition while incrementing accounter :** This iteration construct is the statement whose head specifies a variable that serves as a counter, or index during the iteration. An initial value, final value, and increment are specified in the head, and the body is executed repeatedly using first the initial value a the value of the index variable, then the initial value plus the increment, then the initial value plus twice the increment, and so on, until the final value is reached. The syntax for Pascal **for** statement is

for < name > : = < expression 1 > **to** < expression 2 > **do** <statement

e.g. **for** $i = 1$ to limit do $A[i] = 0$

Here after each iteration, the value of i is increment by 1.

The alternative form of the **for** statement in Pascal is

for < name > : = < expression 1 > **down to** < expression 2 > **do** < statement >

The variable is decrement for the next execution of the statement.

The syntax of a **for** statement in C is given as

for (expression 1 ; expression 2 ; expression 3)
loop body

The loop body can be a single statement, a compound statement, or a null statement. Also,

expression 1 → Used for initialization and is evaluated only once.

expression 2 → Used for loop control and is evaluated before each execution of the loop body.

expression 3 → Used to increment the loop counter and is executed after each execution of the loop body.

Therefore, the design of **for** statements in a language depends on the treatment of:

- The **index** variable, which controls the flow through the loop.
- The **step,** which determines the value added to the index variable each time through the loop.
- The **limit,** which determines when control leaves the loop.

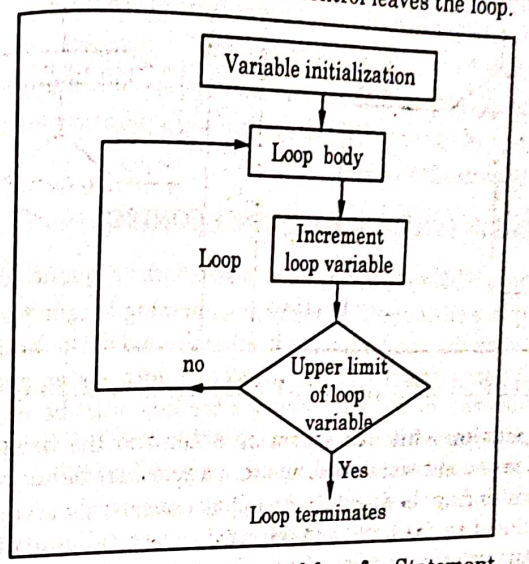


Fig. 6.12. The Flow of Control for a **for** Statement.

The flow of control for a **for** statement is shown in figure 6.12.

(b) **Indefinition Repetition :** Where the conditions for loop exit are complex and the number of executions are not known, the number is known by the course of the computation. *e.g.* in Ada

```
loop
   ...
   exit when condition
   .....
   end loop ;
```

The flow of control is shown in figure 6.13.

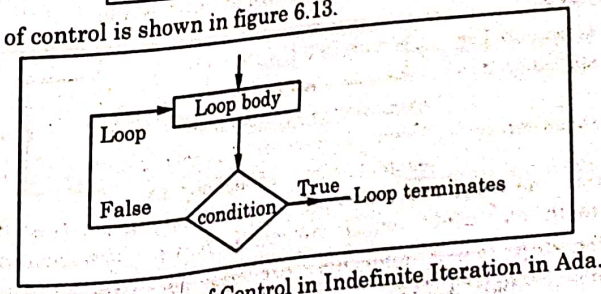


Fig. 6.13 The Flow of Control in Indefinite Iteration in Ada.

Implementation : The implementation of loop control statements using the hardware branch / jump instruction is straight forward. To implement a **for** loop, the expressions in the loop need defining the final value and increment must be evaluated on initial entry to the loop and saved in special temporary storage areas where they may be retrieved at the beginning of each iteration for use in testing and incrementing the controlled variable.

6.5.5 PROBLEMS IN STRUCTURED SEQUENCE CONTROL

The various problems that may arise in structured sequence control are :

1. **Multiple exit loops :** In many programming languages, a situation may arise when the loop terminates if either the end of the limit is reached or an appropriate element is found, therefore a **goto** statement must be used to exit from the middle of the loop or a **for** loop must be replaced by a **while** loop. However, **the exit** statement in Ada and the **break** statement in C provides an alternative without use of a goto statement.
2. **do-while-do :** A **do-while-do** loop is executed as many times as needed and then only the first half is executed on last iteration (due to the exit test present in the middle). Again the **exit** statement in Ada dn **break** statement in C provides on alternative. The do-while-do construct is also called "**loop-and-a half**".
3. **Exceptional conditions :** In the programming languages that do not implement special exception handling statements, a **goto** statement is used. However Ada provides a **raise** statement for exception handling.

6.5.6 PRIME PROGRAMS

The theory of prime programs can be used to describe a consistent theory of control structures. The prime program was developed by Maddux as a generalization of structured programming to define the unique hierarchical decomposition of a flowchart. Any flowchart is a graph of directed ares and these three types of nodes. Consider 3 classes of flow chart nodes as shown in figure 6.14.

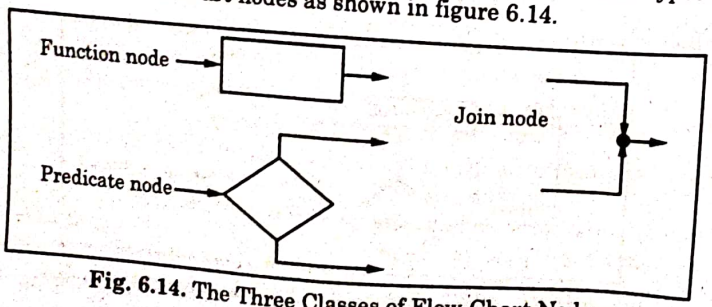


Fig. 6.14. The Three Classes of Flow Chart Nodes

(i) **Function node :** A function node represent computations by a program and represents an assignment statement resulting in change in the state of virtual machine after the execution of the statement.

(ii) **Predicate node :** Also called Decision node, these represents predicates and control flows out of a decision box on either the true or false branch.

(iii) **Join node :** A join node is represented as a point where two arcs flow together to form a single output are

Proper Program : A proper program is a flow chart with

(i) Single entry arc

(ii) Single exit arc

(iii) There is a path from entry arc to any node to exit and vice-versa.

Prime Program : A prime program is a proper program that cannot be subdivided into smaller proper programs. *i.e.* a prime program has no embedded proper subprogram of greater than one node, *i.e.* cannot cut two arcs to extract a prime subprogram with in it.

Composite program : A composite program is a proper program that is not prime.

6.6.1 ADVANTAGES OF PRIME PROGRAMS

The prime programs has the following advantages.

1. Prime programs are easy to understand.
2. The changes to the state space by executing these primes becomes manageable.
3. All primes can be enumerated.
4. The decomposition of every proper program into a hierarchical set of prime subprogram is unique except for special case of linear sequence of function nodes.
5. The structure theorem reflects the fact that any prime program can be built out of structure control statements (*i.e.* only **while** and **if** statements) that provides the advantages of structured programming.

6 SUBPROGRAM SEQUENCE CONTROL

The sequence control mechanism at subprogram level controls the sequencing between two subprograms, how one subprogram invokes another and the called program returns to the first. The subprogram call and return operation of a are together called its **subprogram linkage.** Any implementation method

for subprograms must be based on the semantics of subprogram linkage. A few points that must be remembered in subprogram sequence control are :

1. A subprogram has a single entry point.
2. The caller is suspended during execution of the called subprogram.
3. The control always returns to the caller when the called subprogram execution terminates.

We can use a subprogram to group them together into a logical unit. As an example of subprogram linkage consider a sort program that calls a swap procedure to swap the values of two variables. The diagram in figure 6.15 shows how the flow of control might loop in sorting program.

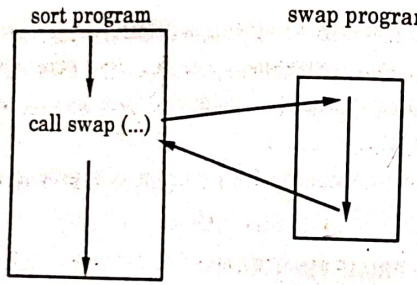


Fig. 6.15. The Flow of Control in Subprogram Linkage.

The flow of control changes as :

(*i*) At the start of the call, the flow of control is transferred to the swap procedure.

(*ii*) Once the **swap** procedure completes, the flow of control transfers back to the calling program.

Note that the call to **swap** procedure includes some data that is not displayed in the diagram. These data are called parameters and are represented by three dots in parenthesis (Figure 6.15). In this section we consider the subprogram sequence control for

(*i*) Simple subprogram call return
(*ii*) Recursive subprograms

6.6.1 SIMPLE SUBPROGRAM CALL-RETURN

Specification : The concept of subprogram linkage described above can be explained by copy rule view of subprograms which states that the effect of a subprogram call statement is the same as if the subprogram were copied into the main program. However, there are some implicit assumptions present in this view of subprogram which are given as follows :

1. Subprograms cannot be recursive because in that case the substitution of subprogram call for subprogram body is unending.
2. Explicit **call** statements are required for the copy rule to apply.
3. Subprograms must execute completely at each call from its beginning to end.
4. At the point of subprogram call, the control is immediately transferred to the called subprogram.
5. Single execution sequence from calling to called subprogram and back to calling program.

The term **"simple"** in simple subprogram call-return means that subprograms cannot be nested and all local variables are static. The execution of the expression or statement sequence means executing a block of code but for subprogram we should need to understand following concept.

(*a*) For the subprogram CALL, the run time system needs to
  (*i*) Save the execution status of the caller.
  (*ii*) Carry out the parameter – passing process.
  (*iii*) Pass the return address to the caller.
  (*iv*) Transfer the control to the caller.

(*b*) For the subprogram RETURN, the system need following actions.
  (*i*) If pass-by-value-result parameters are used, move the current values of those parameters to their actual parameters.
  (*ii*) If it is a function, move functional value to a place the caller can get it.
  (*iii*) Restore the execution status of the caller.
  (*iv*) Transfer the control to the caller.

The simple flow of execution in simple subprogram call-return is as shown in figure 6.16.

Implementation : For simple subprogram call-return implementation we need to recall the following points.

1. The subprogram definition is different from subprogram activation in that the definition is translated into a template which is used to create an activation each time a subprogram is called.

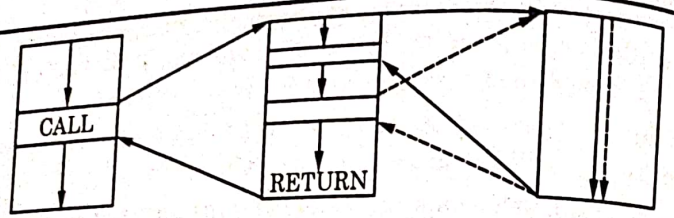


Fig. 6.16 Simple Subprogram call-return.

2. The subprogram activation consists of
   (*a*) The invariant part called cage segment consisting of executable code and constant.
   (*b*) The dynamic part called an activation record consisting of local data, parameter etc.

   The code segment and an activation record are shown in figure 6.17.
3. Each time a subprogram is called a new activation record is created and on subprogram return the activation record is destroyed.
4. The execution is implemented by the use of two system-defined pointers.

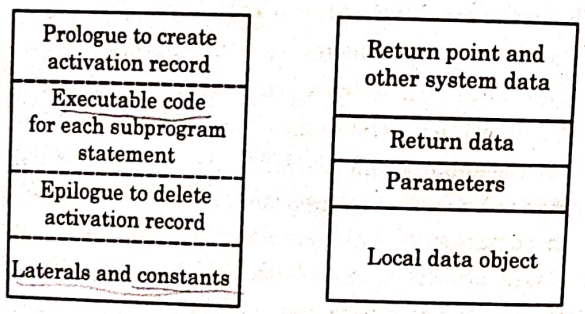


Fig. 6.17 A Subprogram Activation : (*a*) Code Segment (*b*) Activation Record.

(*a*) Current-instruction pointer (CIP) : At any point during execution, the instruction in code segment that us current being executed (or just about to be executed) is called the current instruction and the address of the next instruction to be executed is maintained in current-instruction pointer (CIP).

(*b*) Current-Environment Pointer (CEP) : More than one activations of same subprogram may exist during execution, therefore a pointer to the current activation record is also needed. The pointer to the current activation record is termed as current environment pointer (CEP).

With the CIP and CEP pointers the program execution is explained as given below :

1. On subprogram call instruction
   - An activation record id created.
   - Current CIP and CEP are saved in the created activation record as return point **(ip, ep).**
   - CEP is assigned the address of the activation record.
   - CIP gets the address of the first instruction in the code segment.
   - The execution continues from the address in CIP.
2. On subprogram return instruction
   - The old values of CIP and CEP are retrieved.
   - The execution continues from the address in CIP.

The modes for the implementation of subprogram call return imposes a restriction that at must one activation of any subprogram is being enacted at any point during program execution. From this property, a simpler model of subprogram implementation may be derived which allocate storage for a single activation record statically as an extension of the code segment used in **FORTRAN** and **COBOL**. The activation record is not destroyed-only reinitialized for each subprogram execution. With the more general implementation of call and return, the underlying hardware provides support in which CIP is represented as the program counter, CEP is not used and a simple jump instruction is executed on return. An example of this structure is shown in figure 6.18.

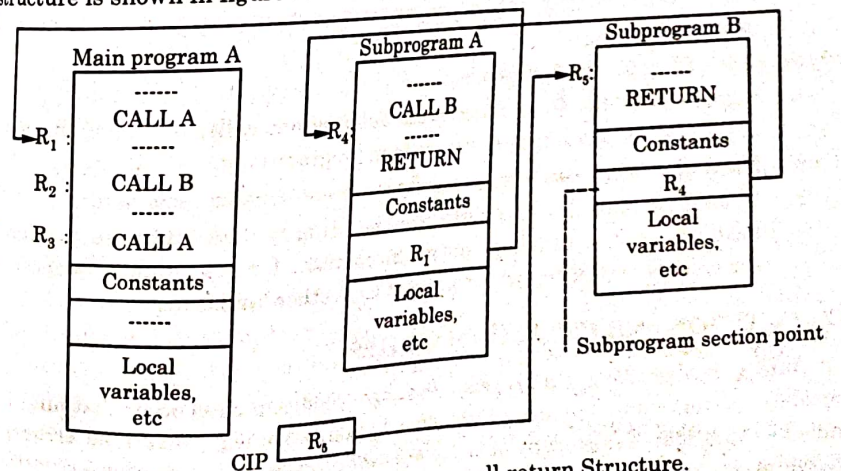


Fig 6.18 Subprogram call-return Structure.

6.6.2 RECURSIVE SUBPROGRAMS

Specification : A subprogram is directly recursive if it calls itself *e.g.* a subprogram A containing a statement call A. Alternatively, a subprogram is indirectly recursive if it calls another subprogram that calls the original subprogram.

Syntactically there is no difference between a simple and recursive subprogram call-return. However, semantically, multiple activations of the same subprogram exist simultaneously at some point in the execution *e.g.* the first recursive call creates a second activation with in the lifetime of the first activation, the second activation may create the third activation and so on. The subprogram activation are destroyed in the reverse order in which those activations are created (*i.e.* stack or LIFO structure).

Implementation : Because of the possibility of multiple activation, we need both the CIP and CEP pointers. The activations are destroyed in the reverse order of activation creation. Therefore recursive subprograms are implemented using the stack based implementation in which

1. CIP and CEP are stored as (ip, ep) in stack, forming a dynamic chain of links.
2. A new activation record is created for each call and destroyed on return.
3. The lifetimes of the activation records cannot overlap-they are nested. Some language compiler (C, Pascal) always assume recursive structure of subprograms, while in others non-recursive subprograms are implemented in the simple way.

Advantages of Recursive Subprograms :

1. Recursion, in the form of recursive subprogram calls, is one of the most important sequence control structure in programming.
2. Many algorithms are most naturally represented using recursion.
3. In LISP, where list structures are the primary data structure available, recursion is the primary control mechanism for repeating sequences of statements, replacing the iteration of most other languages.

6.7 EXCEPTION AND EXCEPTION HANDLERS

During the execution of a program, events or condition often occur that might be considered exceptional. The errors detected by hardware (*e.g.* disk read errors) or unusual conditions detected by a software interpreter are termed as exceptions. Therefore, we defined an **exception** to be any unusual event, erroneous or not, that

detectable either by hardware or software and that may require special processing.

The special processing that may be required when an exception is detected is called **exception Handling** and the subprogram that performs the special processing (*i.e.* exception handling) is termed as **exception handler.**

The action of noticing the exception, interrupting program execution and transferring control to the exception handler is called **raising the exception.** An exception is raised when its associated event occurs.

The process of exception-handling control flow is shown in figure 6.19.

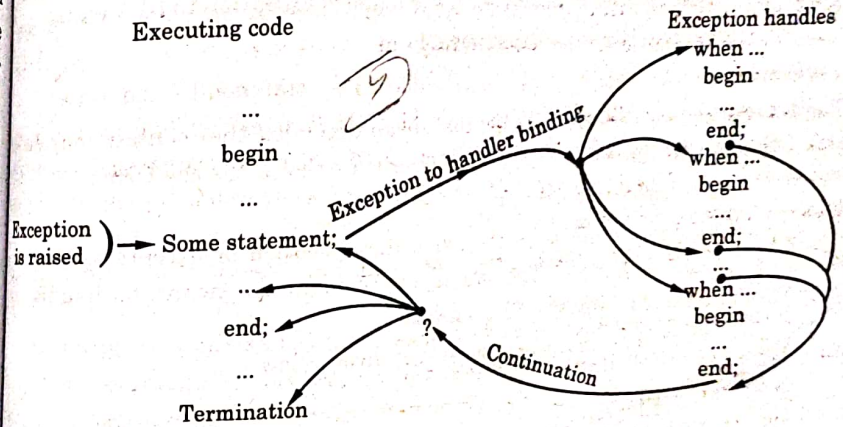


Fig 6.19 Exception-handling Control Flow.

There are following design issues involved in exception handling

1. **Binding an exception to an exception handler :** When there is no exception handler local to the unit in which the exception is raised, the designer must decide whether to propagate the exception to some other unit. On the other hand if exception are propagated, a single handler might handle the same exception raised in general program units, which may require the handler to be more general than one would prefer.
2. **Control Continuation :** After an exception handler executes, either control can transfer to some where in the program outside of the handler code, or program execution can simply terminate. This process is termed as control continuation.

Exception Handler : An exception handler is defined as the unit or segment that performs the special processing to handle the exceptions. An exception handler

is invoked without an explicit call. The definition of an exception handler typically contains

(*i*) A set of declarations of local variables, if any, and

(*ii*) A sequence of executable statements

Note here that an exception may be

(*a*) Language defined *e.g.* Program-Error, Constraint-Error, Numeric – Error in Ada.

(*b*) Programmer defined as a declaration like "Underflow : **exception**"

Each exception handler is paired with the name of exception to be handled. The exception handler have the general form

when exception – choice { | exception – choice} ⇒ statement – sequence

where the braces are metasymbols that mean that what they contain may be left out or repeated any number of times. The exception – choice has the form :

exception-name/**others**

The exception-name indicates the particular exception or exceptions that this handler is meant to handle. The statement sequence is meant to handle any exceptions not named in any other local handler.

The usual form of an exception clause is as given below.

```
begin
... the block or unit body ...
exception
    when exception-name-1 ⇒
        ..... first handler .....
    when exception-name-2 ⇒
        ... second handler .....
        ... other handlers ...
end ;
```

C ++ uses a special construct called **try** clause for exception handling. A try construct includes a compound statement called the **try** clause and a list of exception handlers. The compound statement defines the scope of the following handlers. The general form of this construct is


```
try {
// * * Code that is expected to raise an exception
}
catch (formal parameter) {
// * * A handler body
}
.....
catch (formal parameter) {
// * * A handler body
}
```

Each of the **catch** functions are exception handlers. A **catch** function can have only a single formal parameter, which is similar to a formal parameter in a function definition in C + +, including the possibility of it being an ellipsis (...). A handler with an ellipsis formal parameter is the catch-all handler; it is enacted for any raised exception if no appropriate handler has been found. In C + +, exception handler can include any C + + code.

Raising an exception : An exception may be raised by a language defined primitive operations or by the programmer using a statement provided for that purpose. *e.g.* in Ada.

raise exception – name ;

may be used to raise an exception.

However in C + +, exceptions are raised only by the explicit statement **throw**, whose general form is

throw [expression] ;

The brackets here are metasymbols used to specify that the expression is optional. A **throw** without an operand can only appear in a handler. When it appears there, it reraises the exception, which is then handled elsewhere. This is exactly like the Ada use of a **raise** statement without an exception name.

Propagating an exception : Often in constructing a program, the place at which an exception occurs is not the best place to handle it when an exception is handled in a subprogram other than the subprogram in which it is raised, the exception is said to be propagated from the point at which it is raised to the point at which it is handled.

Advantages : There are some definite advantages to having exceptio handling built into a language. These are as follows :

1. The presence of exception handling in the language could permit th compiler to insert checks in the code when requested by the program.
2. Without exception handling the code required to detect error conditions co clutter a program.
3. Exception propagation provides an advantage of raising an exception in on program unit while handling it in another program unit in its dynamic o static ancestry.
4. The presence of exception handling simplifies the dealing with unusu situations.
5. The reuse of a single exception handler for a large number of differen program units results in significant savings in development cost an program size.

6.8 COROUTINES

A coroutine is a special subprogram that has multiple entries. They can b used to provide interleaved execution of subprograms. In general terms, a coroutine is a special subprogram which returns to their calling program befor the completion of execution. A continue executes partially, then execution i suspend, control is returned back to caller and execution is resumed later fro the point of suspension.

The first high-level language to include the facility for coroutines wa SIMULA 67. Other languages that support coroutines are BLISS, INTER LISP and MODULA-2. The name coroutine derives from the symmetry. rather tha having a parent-child or caller-callee relationship between the two subprograms the two programs appear more as equals-two subprograms swapping control back and forth as each executes, with neither clearly controlling the other.

Typically, coroutines are created in an application by a program unit called the **master unit** which is not a coroutine. When created, coroutines execute their initialization code and then return control to that master unit. The figure 6.20 shows the execution of coroutine A : started by master unit.

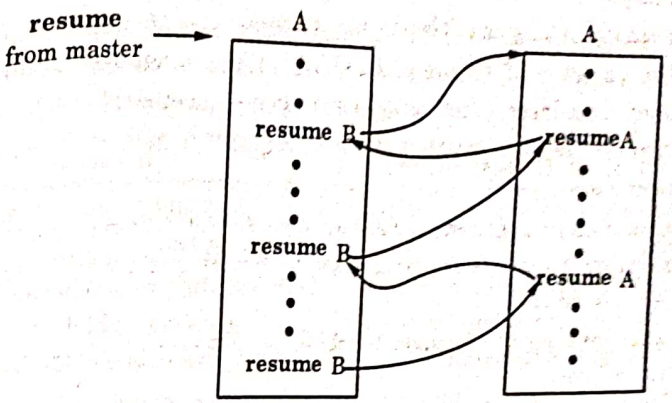


Fig. 6.20 Control Transfer Between Coroutines.

After some execution A, starts B when coroutine B, first causes control to return to coroutine A, the semantics is that A continues from where it ended its last execution. In particular its local variables have the values left them by the previous activation.

Implementation : The assumption that there is at most one activation of a given coroutine existing at time allows us to use an implementation for coroutines similar to that used for simple subprogram call-return in section 6.6.1. The **resume** instruction that transfer the control between coroutines specifies the resumption of some particular activation of the coroutine. A single activation record is allocated storage in the beginning. The resume point is reserved in activation record to save the old ip value of CIP when a **resume** instruction transfers control to another coroutine. The execution of a **resume** B instruction in coroutine A involves follows two step :

(*i*) The current value of CIP is saved in resume point location of the activation record for A.

(*ii*) The **ip** value in the resume point location of B is fetched from B's activation record and assigned to the CIP to effect the transfer of control to the proper instruction in B.

Because there is no explicit return instruction, B does not need to know that A gave it control. The control structures involving recursive coroutines have more complex implementations.

6.8.1 SUBPROGRAMS AND COROUTINES

A coroutine maintains one of the usual characteristics of subprograms *i.e.* only one coroutine actually executes at a given time. However coroutine differs from a subprogram (or subroutine) in following points summarized in table 6.6.

Subprogram (or subroutine)	Coroutine
1. A subprogram executes completely when called.	1. A coroutines executes partially when called.
2. Each time a subprogram is called, it begins execution from the beginning of subprogram.	2. A coroutine resumes execution from the point at which the execution was previously suspended.
3. There is a master-slave relationship between a caller and callee subprogram.	3. Both caller and callee routines are treated on equal basis and no master-slave relationship is present.
4. A subprogram has a single entry point.	4. A coroutine may have multiple entry point.
5. A subprograms is not history sensitive.	5. A coroutine must be history sensitive.
6. The invocation of a subprogram is called a subprogram **call.**	6. The invocation of a coroutine is called coroutine **resume.**
7. It provide the complete execution.	7. It provides interleaved execution.

Table 6.6 : Differences between subprogram and coroutine

KEY POINTS TO REMEMBER

- A **sequence control** is defined as the control of the order of execution of operations, both primitive and user defined.
- A **sequence control** may be either **implicit** or **explicit.**
- **Implicit sequence control** structures are those defined by the language and is determined by the order of the statements in the source language.
- **Explicit sequence control** structures are those defined by the programmer to modify the implicit sequence control defined by the language.

- The three different levels at which the sequence control mechanisms may be applied are expressions, statements and subprograms.
- The sequence control mechanisms within expression consider :
  - — Sequencing with arithmetic expressions
  - — Sequencing with non-arithmetic expressions
- The linear representation of the expression free provides the prefix, postfix and infix notations for writing the syntax of expressions.
- The semantics for expression evaluation can be interpreted by using the implicit control rules of
  - — Operator precedence (*i.e.* Hierarchy of operations)
  - — Associativity of operators.
- The sequencing with non-arithmetic expressions involves pattern matching, unification on Backtracking etc.
- The three main forms of statement-level sequence control are **composition, alteration** and **iteration.**
- The **explicit sequence control** between statements can be provided by goto statements, break and continue statements.
- A **structured program design** uses single-entry, single-exit structure and used three control structures *i.e.* compound statements, conditions and Iterations.
- A **proper program** is a flow chart with single entry arc, single exit arc. Also, there is a path from entry arc to any node to exit and vice versa.
- A **prime program** is a proper program that cannot be subdivided into smaller proper programs.
- A **composite program** is a proper program that is not prime.
- The **sequence control mechanism** at subprogram level controls the sequencing between two subprograms, how one subprogram invokes another and the called subprogram return, to the first.
- The subprogram call and return operations of a language are together called its subprogram linkage.
- The subprogram sequence control can be simple subprogram call return or recursive subprograms.
- The simple call-return mechanism uses the two system-defined pointers : **CIP (current-instruction pointer)** and **CEP (current environment pointer).**
- The simple call-return mechanism uses the two system-defined pointers : CIP (current -instruction pointer) and EEP current environment pointer)

- A subprogram is **directly recursive** if it calls itself. if it call anot subprogram.
- A push down automata (PDA) is an abstract machine similar to fin state automata but in addition it has a stack. .
- The **static semantics** are described using more powerful mechanis called attribute grammer.
- **Attributes** can be synthesized or inherited.
- The **dynamic semantics** includes operational, denotational a axiomatic semantics, algebraic, and translation semantics.

EXERICISE

1. What is a sequence control ? Differentiate between implicit and expli sequence control.
2. Explain the sequence control for arithmetic and non-arithme expressions in detail.
3. Explain the sequence control within statements.
4. What is structured sequence control ? Explain the advantages a disadvantages of structural sequence control.
5. What is a subprogram sequence control ? Explain simple subprogram c return and recursive subprograms in detail.
6. What is an exception ? Explain the concept of exception handling in deta
7. Write short notes on :
   (*a*) Substitution and unification
   (*b*) Proper, prime and composite program
   (*c*) CIP and CEP
   (*d*) Subprogram and co-routine.

CHAPTER

7 CONCURRENCY

7.1 INTRODUCTION

It computer science, concurrency is a property of systems in which several computational processes are executing at the same time, and potentially interacting with each other. The study of concurrency emcompasses a broad range of systems, from tightly-coupled, largely synchronous parallel computing systems, to loosely coupled, largely asynchronous distributed systems. Some people consider that the terms **concurrency** and **parallelism** are same. But in general, the term **parallel** implies the position of multiple processors, while **concurrent** suggests that processes are running at the same time. The concurrent processes may be executing truly simultaneously, in the case that they run on separate processors, or their execution steps may be interleaved to produce the appearance of concurrency, as in the case of separate processors running on a multitasking system. Because the processes in a concurrent system can interact with each other while they are executing, the number of possible execution paths in the system can be extremely large, and the resulting behaviour can be very complex. The various concepts regarding concurrency, categories of concurrency and synchronization methods are discussed in this chapter.

7.2 CONCURRENCY

The fundamental concept of **concurrent programming** is the notion of process (or **tasks**). The **concurrent computing** is simultaneous execution of multiple interacting computation tasks.

A **task** (or **process**) is a unit of program that can be in concurrent execution with other units of the same program. Each task in a program can provide one thread of control. The **thread** of a sequential computation is the sequence of program points that are reached as control flows through the source text of the program.

The tasks may be implemented as separate programs, or as a set of processes or threads created by a single program. The tasks may be executing on a single processor, several processors in close proximity, or distributed across a network.

Concurrent computing is related to parallel computing, but focuses more on the interaction between tasks. Correct sequencing of the interactions or communications between different tasks, and the coordination of access to resources that are shared between tasks, are key concerns during the design of concurrent computing systems. The interaction between processes take two forms.

(i) Communication involves the exchange of data between processes, either by an explicit message or through the value of shared variables. A variable is shared between processes if it is visible to the code for the processes.

(ii) Synchronization relates the thread of one process with that of another. If p is a point in the thread of a process P, and q is a point in the thread of a process Q, then synchronization can be used to constrain the order in which P reaches p and Q reaches q. In other words, synchronization involves the exchange of control information between processes.

7.2.1 MOTIVATION AND ADVANTAGES

There are at least two reasons to study concurrency.

1. To provide a method of conceptualizing program solutions to problems.
2. The need for software to make effective use of the increasing popularity of the hardware capability to support multiple processor concurrency.

In addition to above motivations the concurrent computing provides the following advantages.

1. **Increased application throughput :** the number of tasks done in certain time period will increase.
2. **High responsiveness for Input/Output :** Input/output intensive applications mostly wait for input or output operations to complete. Concurrent programming allows the time that would be spent waiting to be used for another task.
3. **More appropriate program struct :** Some problems and problem domains are well-suited to representation as concurrent tasks or processes.
4. **Increase in programming flexibility :** Concurrent control methods increase programming flexibility.

7.2.2 CONCURRENCY AS INTERLEAVING

The concurrent computations are described in terms of events, where an event is an uninterruptible action and it might be the execution of an assignment statement, a procedure call, the evaluation of an expression in short anything the language in

question chooses to treat as atomic. The thread of a process corresponds to a sequence of events. A thread of control in a program is the sequence of program points reached as control flows through the program.

Interleaving of threads is a convenient technical device for studying the concurrent execution of processes. An interleaving of two sequences s and t is any sequence u formed from the events of s and t, subject to the following constraint : the events of s retain their order in u and so do the events of t.

Interleaving is based on the assumption that concurrent programs can be characterized by the relative order of events. If a and z are concurrent events, then we consider the case in which a occurs before z and the case in which z occurs before a but ignore the case in which a and z occur at the same time. As the number of events on a thread increases, the number of possible interleaving grows rapidly.

As an example, consider that the execution of a process A consists of two events a b and an execution of a process Z consists of three events x y z, then concurrent execution of A and Z can be studied by considering the following possible interleavings :

$a\ b\ x\ y\ z$
$a\ x\ b\ y\ z$
.....
$x\ y\ z\ a\ b$

The interleaving preserves the relative order of events in a thread, so a must occur before b, but x, y and z, being on a separate concurrent thread, can occur in any order relative to a and b. Similarly, x must occur before y and y must occur before Z, but a and b, being on a separate thread, can occur in any order relative to x, y and z.

The 10 possible interleavings of the threads of A and Z are illustrated in figure 7.1.

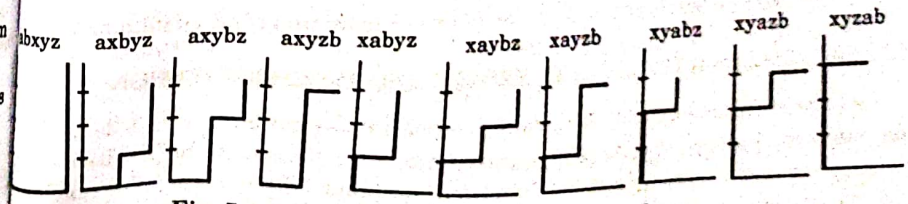


Fig. 7.1. Interleaving at the Threads ab and xyz.

The solid lines in each diagram have a horizontal step for an event in the thread of A and a vertical step for a thread in the event of B. Thus, the interleaving of abc is represented by two horizontal steps followed by three vertical steps. Although interleaving generalizes to any number of tasks, two-dimensional diagrams can be drawn only for interleaving of two processes.

7.3 CATEGORIES OF CONCURRENCY

There are two distinct categories of concurrent unit control :

(*i*) Physical concurrency

(*ii*) Logical concurrency

(*i*) Physical Concurrency : The physical concurrency occurs when several program units from the same program literally execute simultaneously on more than one processor. It indicates the presence of multiple independent processors (multiple threads of control).

(*ii*) Logical Concurrency : The logical concurrency occurs when the execution of several programs takes place in an interleaving fashion in a single processor.

7.4 CONCURRENCY LEVELS

The concurrency can be divided into following levels.

1. **Instruction level** is the execution of two or more machine instructions simultaneously.
2. **Statement level** is the execution of two or more statements simultaneously.
3. **Unit level or subprogram level,** is the execution of two or more subprogram units simultaneously.
4. **Program level,** is the execution of two or more programs simultaneously.

7.5 SUBPROGRAM LEVEL CONCURRENCY AND SYNCHRONIZATION

As described in section 7.4, the **subprogram level concurrency** is defined as the execution of two or more subprogram units simultaneously. Before discussing subprogram level concurrency in detail we introduce the concept of a task (or process) in detail.

A **task** is a program unit that can be in concurrent execution with other program units. Tasks differ from ordinary subprograms in that :

(*i*) A task may be implicitly started.

(*ii*) When a program unit starts the execution of a task, it is not necessarily suspended.

(*iii*) When a task's execution is completed, control may not return to the caller.

(*iv*) Task execution control is maintained by a program called the scheduler, which maps task execution onto available processors.

(*v*) Tasks can be in one of several different states.

- **New :** Created but not yet started.
- **Runnable or ready :** Ready to run but not currently running (no available processor)
- **Running**
- **Blocked :** Has been running, but cannot now continue (usually waiting for some event to occur)
- **Dead :** No longer active in any sense.

A task can communicate with other tasks through shared nonlocal variables, ough message passing, or through parameters. During the concurrent execution several tasks, each task may proceed asynchronously with the other *i.e.*, each k executes at its own speed, independently of the others. For example, when task has executed Z statements, task B which was initiated at the same time, may have ecuted only 5 statements or no statements, or it may have already run to mpletion and terminated.

Also, the tasks often work together to create simulations or solve problems and erefore are not disjoint. Therefore, for two tasks, running asynchronously to ordinate their activities, the language must provide a means of synchronization, so at one task can tell the other when it completes execution of a particular section of code. The tasks may use signals to synchronize their activities. The signals serve tell each task when to wait and when to proceed.

In general terms, synchronization is a mechanism that controls the order in hich tasks execute. Two kinds of synchronization required when tasks share data are

These are described as given below :

(*a*) Cooperation synchronization

(*b*) Competition Synchronization

These are described as given below :

(*a*) Cooperation synchronization : The **cooperation synchronization** between two tasks A and B is required when task A must wait for task B to complete some specify activity before task A can continue its execution *e.g.*

in producer-consumer problem, the producer puts data (or values) into buffer at some rate, and the consumer gets data (or values) out of the buffer at some possibly different rate. The sequence of stores to and removals from the buffer must be synchronized. The consumer unit cannot remove data from the buffer if the buffer is empty. Also, the producer cannot place data in buffer if it is full. This is called the problem of cooperation synchronization because the users of the shared data structure must cooperate if the buffer is to be used correctly.

(*b*) **Competition synchronization :** The competition synchronization between two tasks A and B is required when both require the use of some resource that cannot be simultaneously used *e.g.* if task A needs access to shared data location x while task B is accessing x, shared counter *i.e.* task A must wait for task B to complete its processing of x, regardless of what that processing is. The competition is usually provided by mutually exclusive access.

7.6 METHODS FOR SYNCHRONIZATION

The method of providing mutually exclusive access to a shared resource is to consider the resource to be something that a task can possess and then follow only a single task to process it at a time. To gain possession of a shared resource, a task must request it. When a task is finished with a shared resource that it possesses, it must relinquish that resource so that the resource can be made available to other tasks. The three methods for providing for mutually exclusive access to a shared resource are :

(*i*) Semaphores
(*ii*) Monitors
(*iii*) Message Passing

These methods are described as given below :

7.6.1 SEMAPHORES

The mechanism of semaphores was devised by Edsger Dijkstra in 1965. A semaphore is a very simple mechanism that can be used to provide synchronization of tasks. The concept of semaphore was first implemented in ALGOL 68. A semaphore acts very much like its namesake on the railroad. When it is down, execution is halted, and when it is up, a process may proceed. The actual workings of a semaphore, including its list of waiting processes, is usually implemented down in an operating system, with users accessing it through its two operations, wait and signal. A semaphore is a data structure consisting of two parts :

(*i*) **An integer counter,** whose value is always positive or zero, that is used to count the number of signals sents but not yet received, and

(*ii*) **A queue,** that stores task descriptors. A task descriptor is a data structure storing a queue of tasks that are waiting for signals to be sent.

Note : In a general semaphore, the integer counter may assume any positive integer value. However, if a counter variable may have only the values 0 and 1 then it is called a **binary semaphore.**

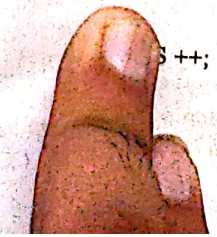
The two primitive operations defined on a semaphore data object S are ;

(*i*) **Wait (S) :** The **wait** operation is sometimes known as **down** or **P**. When executed by a task A, this operation tests the value of the counter in S ; if non zero, then the counter value is decremented by one indicating that the task has received a signal and the task continues execution. If zero, then task is inserted at the end of the task queue for S and execution of task is suspended indicating that B is waiting for a signal to be sent. The implementation of **wait (S)** is given as

```
int S ;
wait (S)
{
   while (S = = 0) {/* no-op */}
   S – –;
}
```

(*ii*) **Signal (S) :** The **signal** operation is sometimes known as **up** or **V**. When executed by a task A, the counter is incremented by one indicating that a signal has been sent but not yet received. However, if the value of counter variable is zero, then the first task from task queue is removed and its execution is resumed. In either case, execution of task A continues after the **signal** operation is complete. The implementation of **signal (S)** is given as

```
int S ;
signal (S)
   S ++;
```



7.6.1.1 USING SEMAPHORES

A semaphores can be used to provide mutual exclusion. The **wait** and **signal** operations have simple semantics that require the principle of **atomicity**, each operation completes execution before any other concurrent operation can access it data.

A **critical section** in a process is a portion or section of code that must be treated as an atomic event. Two critical sections are said to be mutually exclusive because their executions must not overlap.

A concurrent program with critical sections is **safe** if it executes the critical sections contiguously, without interleaving. The two cyclic process in figure 7.2. are allowed to execute their critical sections in any order, even if P executes more often than Q.

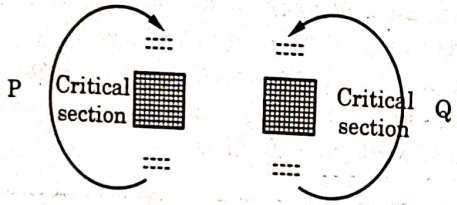


Fig. 7.2 Two Cyclic Processes with Critical Sections.

The use of semaphores to provide such mutual exclusion is shown as below.

```
int mutex=1;

while (TRUE)
{
   wait (mutex);
   critical section
   signal (mutex);
   non critical section
}
```

This is known as a **mutex** block

Fig. 7.3 Using Semaphore for Providing Mutual Exclusion.

Figure 7.3 : using Semaphores for providing mutual exclusion

A semaphore can also be used to ensure that one section of code is run before another section, even when they are in different processes as shown in figure 7.4.


```
Process: 1
int signal =0;

first section
signal (synch);
```

```
Process 2:

wait (synch)
second section
```

Fig 7.4. Use of Semaphore to Ensure that one Section of Code is Run Before Another Section

7.6.1.2 IMPLEMENTING SEMAPHORES

A typical implementation of a semaphore is given as shown in figure 7.5.

```
typedef struct {
    int value;
    struct process * L;
} semaphore;
```

```
void wait (semaphore S)
  {
      S.value – –;
      if (S.value <0){
          add this process to S.L;
      sleep ( );
  }
  }
```

```
void signal (semaphore S)
  {
      S. value ++;
      if (S. value <=0)
  {
  remove a process P from S.L;
      wakeup(P);
  }
  }
```

Fig 7.5 Implementing of Semaphores.

7.6.1.3 ADVANTAGES AND DISADVANTAGES

The semaphores provides following advantages

1. Semaphores provides cooperation synchronization.
2. Semaphores provide mutual exclusion.
3. Semaphores are simple and powerful.
4. Semaphores avoid busy waiting.

cooperation Sync.
mutual exclusion
Simple & powerful
easy
Avoid busy waiting

5. Semaphores can be used to ensure that one section of code is run before another section.

However, the semaphores provides the following disadvantages :

1. It is really hard to use semaphores correctly.
2. Easy to mix up **wait ( )** and **signal ( )**.
3. Programs involving several tasks and semaphores become increasingly difficult to understand.
4. They are easy to mission figure.
5. Easy to omit **signal ( )**, especially when handling errors.
6. A task can wait for only one semaphore at a time but often it is desirable for tasks to wait for several signals.
7. If a task fails to **signal** at the appropriate point, the entire system of tasks may deadlock.
8. Semaphores are structure oriented.
9. There is no exchange of data.

7.6.2 MONITORS

The critical sections and semaphores represent an early approach to provide exclusive access to shared data. However, the synchronization using semaphores imposes certain problems. Therefore, in 1971, Edseger Dijkstra suggested that all synchronization operations on shared data be gathered into a single program unit. Brinch Hansen formalized this concept in the environment of operating systems. In 1973, Hoare, named these structures monitors.

The first programming language to incorporate monitors was concurrent Pascal. Modula, CSP/K and Mesa also provide monitors. Monitors are a language construct to encapsulate shared data structures with their operations and hide their representations — *i.e.* make shared data structures abstract data types. This solution can provide competition synchronization without semaphores by transferring responsibility for synchronization to the run time system.

A monitor is an interface between concurrent user processes and provides

(*i*) A set of procedures callable by users.

(*ii*) A mechanism for enforcing mutual exclusion on those procedures — only one process can be executing (or in the ready queue) with its program counter in monitor code.

(*iii*) A mechanism for scheduling calls to these procedures if other concurrently executing processes request usage before the procedure has terminated.

(*iv*) A mechanism for suspending a calling procedure until a resource is available (delay) and then reawakening the process (continue).

A monitor has no access to nonlocal variables and can communicate with other monitors only by calling procedures in them. Thus a monitor serves as a third-party policeman between two or more cooperating processes.

A monitor can be considered as an abstract data type that includes a shared data structure and all the operations various concurrent processes can perform on it. These operations determine an initialization operation access rights, and synchronizing operations. Concurrent processes P_1 , P_2 ..... P_n must be prevented from accessing the same data item simultaneously.

A monitor has the form shown in figure 7.6.

```
monitor monitor-name
{
        shared variable declarations
        Procedure P1 (···)
        {
        ...
        }
        ...
        Procedure Pn (···)
        {
        ...
        }
        {
        initialization code
        }
}
```

Fig 7.6 The General Form of a Monitor.

Key points to remember here are

1. Shared variables are accessible through the monitor's procedures.
2. The language performs locking so that only one process can be running any monitor procedure at a time.
3. All the critical sections are put inside monitor procedures.
4. One process can be active in monitor at any time.

7.6.2.1 MONITOR CONDITION VARIABLES AND OPERATIONS

Sometimes a critical section will need to wait for another process to do work, like a consumer encountering an empty queue. Therefore, monitors need a blocking mechanism, which is provided by condition variables. A condition variable is not really a variable. It does not have a value and processes cannot examine it. Condition variables are used to add additional synchronization behaviour beyond basic mutual exclusion. The condition variables are declared as *e.g.*

condition *x* ;

The only operations that can be performed on condition variables are wait( ) and **signal ( )** :

(*i*) ***x* . wait ( )** behaves like **sleep ( )**

(*ii*) ***x* . signal ( )** behaves like **wakeup ( )**, and wakes up the process sleeping on *x*.

In addition to operations defined on monitors, there are two special operations :

***(i)* Delay :** It is analogous to semaphore's wait operation and the execution of a **delay** enqueues a process.

***(ii)* Continue :** It is analogous to semaphore's signal operation and the continue operation dequeues the first waiting process and allow it to enter the monitor.

6.2.2 ADVANTAGES AND DISADVANTAGES

The monitors provides following advantages.

1. A monitor provides interface between concurrent user processes.
2. A monitor allow concurrent processes to prevent from accessing the same data item simultaneously.
3. The main benefit of monitors is in the clarity and reliability of the system using them, not their operation.
4. Monitors encapsulate the shared data structure with their operations.
5. The advantage of monitors is that all the code for the buffer appears together.
6. Monitors are a better way to provide competition synchronization then semaphores.
7. Monitors are a nice way to write multitasking programs.

However, the monitors provide the following disadvantages.

1. The main problem with monitors is that a process may block within a monitor.
2. The monitors cannot provide synchronization of units in a distributed system where each processor has its own memory.
3. Monitors are structure oriented.
4. The monitors need shared memory.
5. Monitors are not even available in some most commonly used languages like C, C++ etc.
6. There is no exchange of data using monitors.

6.3 MESSAGE PASSING

The monitor construct is a dependable and safe method for providing mpetition synchronization for shared data access in concurrent units that share a ngle memory. Monitors and semaphores are basically designed for machines that are a single memory – they are data-structure-oriented. However, if two processes annot share memory, then none of the solutions (monitors and semaphores) escribed can work.

As separate computers become more prevalent, another paradigm, **message assing,** has shown itself to be particularly valuable. The concept of message assing is most appropriate for distributed computing since the shared memory is ot required.

The first efforts to design languages to provide message passing capabilities ere those of Hansen and Hoarse. The message passing mechanism involves a

(*a*) A source, from where a message is sent.

(*b*) A destruction, where the message is supposed top be delivered.

(*c*) A communication channel defined by a source and destination.

The simplest designation is

***(i)* Direct naming :** The direct naming message passing involves sending data to a receiver or receive data from a sender where sender (source) and receiver (destination) are the names of the processes. The two primitives used are

send (destination, & message) ;

receive (source, &message) ;

These primitives are system calls and not the language constructs. The first primitive says that a message is sent to destination. While the second

primitive says that a message is received from source. These two primitives are shown in figure 7.7.

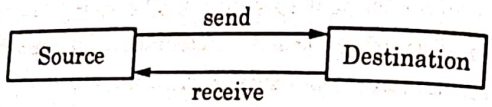


Fig. 7.7 Message Send and Message Receive Primitive.

(ii) **Indirect naming using a buffer,** are sending or receiving messages, buffering may be necessary to hold a message until a receiving process is ready for it. Such a buffer is often called a **mailbox.** In the particular case where there is only one receiver but many senders, the monitor is called a port. The two primitives used are

send (mailbox, & message)

receive (mailbox, & message)

These primitives are system calls, not language constructs. The first primitive says that a message is sent to a buffer called mailbox while the second primitive says that a message is received from a buffer called mailbox. These two primitives are shown in figure 7.8.

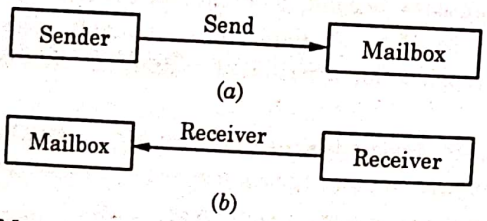


Fig. 7.8 Message **Send** and (*b*) Message **Receive** Primitive.

7.6.3.1 MESSAGE PASSING MODELS

There are four basic message passing models.

1. **Point-to-point :** It is the simplest message passing technique involving one process sending a message to another, which receives it. The point-to-point messages can be passed synchronously asynchronously. The example languages that provide point-to-point message passing are SR and concurrent C etc.
2. **Rendezvous :** The synchronization of actions of two tasks for a brief period is termed as rendezvous. The rendezvous includes synchronization

communication and execution of a block of code in one of two or more concurrently running processes. A rendezvous can occur only if both the sender and receiver want it to happen. The information of the message can be transmitted in either or both directions. The example languages that provide rendezvous are Ada, CSP, concurrent C etc.

3. **Remote procedure calls :** RPCs are much like the processes used to accomplish rendezvous. They are intended, however to, have exactly the same meaning as regular procedures. When this can be achieved, it permits the coding of concurrent processes in traditional procedural languages and allow conventional programs to be posted into the synchronizing system. RPCs have been considered for use with Modula-2 and implemented in the V operating system and in concurrent CLU etc.
4. **One-to-many message passing :** One-to-many message passing is also called **broad casting,** as it behaves much like a radio station, when all receivers hear the same message. This message passing scheme can be either buffered ot unbuffered. One language implementing one-to-many message passing is Broadcasting sequential processes (BSP), another descendent of CSP.

7.6.3.2 ADVANTAGES AND DISADVANTAGES

The message passing method provides following advantages.

1. There is no need to share memory.
2. Message passing is most appropriate for distributed computing. Since shared memory is not required.
3. Message passing is not data-structure oriented.
4. Message passing is a nice paradigm for inter-machine messaging.
5. No critical sections or regions are required to be maintained.
6. Message passing provides sharing of data values through passing the values as messages.
7. Message passing is extensively used by many systems like Mach the basis of Apple's new Macintosh OS.

However, the message passing methods provide the following disadvantages.

1. **Reliability :** If the processes are in different machines, a protocol must ensure that messages are delivered in order and none is lost.
2. **Addressing :** The processes must have a way of naming each other and arranging for messages to be delivered at all.

3. **Authentication :** If processes are on different machines, it is more difficult to ensure that the message came from who it purports to have originated it.
4. **Expensive :** Message passing is more expensive than semaphores and monitors, so often used for communication between processes on remote machines – multi-computer programming.
5. Another major disadvantage is how to designate the sources and destinations and synchronization of processes.

KEY POINTS TO REMEMBER

- The **concurrent computing** is simultaneous execution of multiple interacting computation tasks.
- A **task** (or **process**) is a unit of program that can be in concurrent execution with other units of the same program..
- **Interleaving of threads** is a convenient technical device for studying the concurrent execution of processes.
- Interleaving is based on the assumption that concurrent programs can be characterized by the relative order of events.
- The two distinct categories of concurrent are **physical** and **logical concurrency**.
- The **concurrency** can be divided at instruction level, **statement level**, **unit (or subprogram) level** and **program level)**
- The two kinds of synchronization required when tasks share data are **cooperation synchronization** and **competition synchronization**.
- The methods for synchronization includes **semaphore**, **monitors** and **message passing**.
- A **semaphore** is a data structure consisting of an integer counter and a question.
- The two primitive operations defined on a semaphore data objects are **wait(s)** and **signal (s).**
- **Monitors** are a language construct to encapsulate shared data structures with their operations and hide their representations *i.e.* make shared data structures abstract data types.
- A monitor has no access to non-local variables and can communicate with other monitors only by calling procedures in them.

- The two additional operations defined on monitors are **delay** and **continue.**
- The concept of **message passing** is useful when two processes cannot share memory and involves a source, destination and a communication channel.
- The message passing can be either using direct naming or indirect naming using a buffer (or mailbox).
  - Point to point
  - Rendezvous
  - Remote procedure calls
  - One to-many message passing

  that calls the original subprogram.
- An **exception** is defined to be any unusual event, erroneous or not, that is detestable either by hardware or software and that may require special processing.
- The special processing that may be required when an exception is detected is called **exception handling** and the subprogram that performs the special processing (*i.e.*, exception handling) is termed as an exception handler.
- The action of noticing the exception, interrupting program execution, and transferring control to the exception handler is called **raising the exception.** An exception is raised when its associated events occurs.
- When an exception is handled in a subprogram other than the subprogram in which it is raised, the exception is said to be **propagated** from the point at which it is raised to the point at which it is handled.
- A **coroutine** is a special subprogram that has multiple entries and can be used to provide interleaved execution of subprograms.
- A subprogram has a single entry point while a co-routine may have multiple entry points.

EXERICISE

1. Define the term concurrency. How it is different from parallelism ?
2. Define the term synchronization. Differentiate between cooperation and completion synchronization.

CHAPTER

8 DATA CONTROL

8.1 INTRODUCTION

A program regardless of the language used specifies a set of operations applied on certain data in a certain sequence. The organization of data and operations into complex executable programs involves following two aspects :

(*i*) Sequence control, and

(*ii*) Data control

The term sequence control has already been discussed in detail in chapter. The various aspects of data control are discussed in this chapter.

8.2 DATA CONTROL

The term **data control** is defined as the control of the transmission of data for each operation of a program. The data control features of a programming language are concerned with following aspects :

1. The accessibility of data at different points during program execution.
2. The determination of how data may be provided for each operation, and how a result of one operation may be saved and retrieved for later use as an operand by a subsequent operation.

The term data control differs from sequence control in the following aspects as shown in table 8.1.

Data Control	Sequence Control
1. The control of the transmission of data among the subprograms of a program is termed as **data control**.	1. The control of the order of the execution of operations, both primitive and user defined is termed as **sequence control**.
2. Data control is ruled by the dynamic and static scope rules for an identifier.	2. Sequence control is ruled by notations in expressions and hierarchy of operations.

Data Control	Sequence Control
3. Data object can be made available through two methods. (*i*) Direct Transmission and (*ii*) Transmission through Reference.	3. Sequence control structures may be either explicit or implicit. Implicit sequence control structures are those defined by the language and explicit are those that the programmer may optionally use.
4. Data control structures may be categorized according to the referencing environment of data.	4. Sequence control structures may be conveniently categorized in three groups. (*i*) Structures used in expressions. (*ii*) Structures used between statements, and (*iii*) Structures used between subprograms.
5. Data control is concerned with the binding of identifiers to particular data objects and subprogram.	5. Sequence control is concerned with decoding the instructions and expressions into executable form.

Table 8.1 : Difference between Data control and Sequence control

8.3 NAMES AND REFERENCING ENVIRONMENTS

There are two ways to make a data object available as an operand for an operation.

1. **Direction transmission,** used for data control with in expressions. In direct transmission, a data object computed at one point as the result of an operation may be directly transmitted to another operation as an operand *e.g* in the computation of

$$a = 2 * b + c;$$

The result of multiplication is transmitted directly as an operand of the addition operation.

2. **Referencing through a named object,** used for most data controls outside of expressions. In this method, a data object may be given a name

when it is created, the name may then be used to designate it as an operand of an operation.

8.3.1 NAMES

A **name** is a string of characters used to identify some entity (or data object) in a program. The earliest programming languages used single-character names. Most of the languages today use multi-character names. Some languages such as C + +, do not specify a length limit on names, although implementers of those languages, sometimes do. In a given program, there are following program elements that may be named.

1. Variable names
2. Formal parameters
3. Subprograms
4. Defined types
5. Defined constants
6. Labels
7. Exception names
8. Primitive operations
9. Literal constants

A name in a program may be

(*a*) **Simple name,** that designates an entire data structure *e.g.* an identifier name.

(*b*) **Composite name,** that designates a component of a data structure *e.g.* student [7] . firstname.

8.3.2 ASSOCIATIONS

An **association** is defined as the binding of identifiers (simple name) to particular data objects and subprograms. During program execution :

(*i*) At the beginning of main program, execution, associations are made (variables and subprograms).

(*ii*) During execution of main program, referencing operations (to be defined shortly) are invoked.

(*iii*) A new set of associations is created when a new subprogram is called.

(*iv*) During the execution of a subprogram, referencing operations are also invoked.

(*v*) The set of associations are destroyed when subprogram returns.

(vi) When the control returns back to main program, the main program continues execution as before.

8.3.2.1 VISIBILITY OF ASSOCIATIONS

Associations are visible if they are part of referencing environment, otherwise associations are hidden.

8.3.2.2 DYNAMIC SCOPE OF ASSOCIATIONS

The dynamic scope of an association includes the set of subprogram activations within which the association is visible.

8.3.3 REFERENCING ENVIRONMENTS

For each program or subprogram, a set of identifier associations available for use in referencing during execution is termed as **referencing environment** of the subprogram (or program). The referencing environment of a subprogram is :

(i) Set up when the subprogram activation is created, and is

(ii) Invariant during one activation.

The referencing environment of a subprogram may have several components.

1. **Local Referencing environment :** The local referencing environment of a subprogram is the set of identifier associations available for use in referencing, that may be determined without going outside the subprogram activation. The local referencing environment of a subprogram is the set of associations created on entry to a subprograms and includes.
   (a) Local variables, defined inside the subprograms *i.e.* access to local variables is restricted to subprograms in which they are defined.
   (b) Formal parameters
   (c) Subprograms defined only with in that subprograms.
2. **Non-Local Referencing Environment :** The non-local referencing environment of a subprogram is the set of identifier associations that may be used with in a subprogram but that are not created on entry to it. The non-local variables of a subprogram are those that are visible with in the subprogram but are not locally declared. It can be global or predefined. The non-local referencing environment includes.
   (a) Non-local variables
   (b) Subprograms enclosing given subprogram.
3. **Global referencing environment :** The global referencing environment of a sub-program is the set of identifier associations created at the start of the execution of the main program, available to be used in a subprogram.

The global referencing environment is the part of the non local referencing environment and includes :

(a) Global variables *i.e.* the variables that are visible in all program units.

(b) The subprograms declared in the main program.

4. **Predefined referencing environment :** The predefined referencing environment of a subprogram is set of identifier associations defined directly in the language definition that may be used without explicitly creating them. The example of a predefined environment in Pascal consist of constants such as MAXINT (the maximum integer value) and subprograms such as read, write and sqrt. Any of these predefined identifiers may be given a new association through an explicit program declaration, and thus the predefined association may become hidden for part of the program.

In order to illustrate the concept of local, non-local and global referencing environment consider the following program segment shown in figure 8.1.

```
void main ( )
   {
      float a, b, c ;
      ......
      void sub 1 (float a)
         {
            float d ;
            ......
            void sub 2 (float c)
               {
                  float d ;
                  ......
                  c = c + b ;
                  ......
               }
            ......
            sub 2 (b) ;
            ......
         }
      ......
      sub (a) ;
      ......
   }
```

Fig 8.1 : A Sample Program in C to Illustrate Referencing Environment.

From the figure 8.1, we observe the following referencing environments.

(i) For Sub 2 : (a) Local referencing environment consist of *c*, *d*.

(b) Non-local referencing environment consist of *a*, sub 2 in sub 1, *b*, sub 1 in main.

(ii) For Sub 1 : (a) Local referencing environment consist of a, d, sub 2.
(b) Non-local referencing environment consist of b, c, sub1 in main.

(iii) For main : (a) Local referencing environment consist of a, b, c, sub 1.

8.3.3.1 REFERENCING ENVIRONMENT OF A STATEMENT

The **referencing environment** of a statement is the set of identifier associations (names) that are visible in the statement. The referencing environment of a statement in a static-scoped language is the variables declared in its local scope plus the collection of all variables of its ancestor scopes that are visible. In such a language, the referencing environment of a statement is needed while that statement is being compiled, so code and data structures can be created to allow references to variables from other scopes during run time.

In order to illustrate the referencing environment of statements consider the following Pascal program segments.

```
program main ;
    var a, b : integer ;
    .....
    procedure sub 1 ;
        var x, y : integer ;
        begin {sub 1}
        ..... ⟷ 1
        end ; {sub 1}
    procedure sub 2;
        var x : integer ;
        .....
        procedure sub 3 ;
        var x : integer ;
        begin {sub 3}
        ... ⟷ 2
        end ; {sub 3}
        begin {sub 2}
        ..... ⟷ 3
        end ; {sub 2}
    begin { main }
    ..... ⟷ 4
    end ; { main }
```

Fig. 8.2. A Sample Pascal Program Segment to Illustrate the Referencing Environments

From the figure 8.2, we observe the following referencing environments.

(i) At point 1, the referencing environment consist of x and y of sub 1, a and b of main.

(ii) At point 2, the referencing environment consist of x of sub 3 (x of sub 2 is hidden), a and b of main.

(iii) At point 3, the referencing environment consist of x of sub 2, a and b of main.

(iv) At point 4, the referencing environment consist of a and b of main,.

8.3.4 REFERENCING OPERATIONS AND REFERENCES

Given an identifier and a referencing environment, a **referencing operation** returns the associated data object or subprogram definition *i.e.* a referencing operation finds an appropriate association for an identifier in a given referencing environment. The signatures of a referencing operation is given as :

ref_op : id × ref_environment → data object or subprogram

8.3.4.1 REFERENCES

A **reference** to an identifier involves finding the association in a particular referencing environment. A reference is said to be :

(i) A local reference, if the referencing identified operation finds the association in the local environment.

(ii) A non-local reference, if the referencing operation finds the association in the non-local environment.

(iii) A global reference, if the referencing operation finds the association in the global environment.

8.4 SCOPE

One of the most important factors having an effect on the understanding of variables is **scope**. The **scope** of a program variable is the range of the statements in which the variable is visible. A variable is **visible** in a statement if it can be referenced in that statement.

The scope rules of a language determine how a particular occurrence of a name is associated with a variable. In particular, scope rule determine how references to variables declared outside the currently executing subprogram or block are associated with their declaration and thus their attributes. A complete knowledge of

these rules for a language is therefore essential to our ability to write or read programs in that language.

Sometimes the scope and lifetime of a variable appear to be related. The **lifetime** of a variable is the time during which the variable is bound to a specific memory location. So, the lifetime of a variable begins when it is bound to a specific cell and ends when it is unbound from that cell. In order to differentiate between scope and lifetime, consider a variable that is declared in a Pascal procedure that contains no subprogram calls. The scope of such a variable is from its declaration to the **end** reserved word of the procedure. The lifetime of that variable is the period of time beginning when the procedure is entered and ending when execution of the procedure reaches the **end** ; Although the scope and lifetime of the variable are clearly not the same because static scope is a textual, or spatial, concept where lifetime is a temporal concept, they atleast appear to be related in this case.

In this section we discuss two types of scope rules.

(*i*) Static scope, and

(*ii*) Dynamic scope

These scope rules are described as given below :

8.4.1 STATIC SCOPE

ALGOL 60 introduced the method of binding names to non-local variables, called **static scoping**. The **static scope** of a declaration is the part of the program text where a use of the identifier is a reference to that particular declaration of the identifier. The static scoping is thus named because the scope of a variable can be statically determined, *i.e.* prior to execution.

Static Scope Rule : A **static scope rule** is a rule determining the static scope of a declaration. The static scope rules relate references with declarations of names in the program text. The important of static scope rule lies in recording the information about a variable during translation.

In all common static-scoped languages except C, C++, Java and FORTRAN, subprogram can be nested inside other subprograms, which can create a hierarchy of scopes in a program. When a reference to a variable is found by a compiler for a static scoped language, the attributes of the variable are determined by finding the statement in which it is declared. In static scoped languages with nested subprograms, this process can be thought of in the following way.

Suppose a reference is made to a variable *a* in subprogram sub 1. The correct declaration is found by first searching the declarations of subprogram sub 1. If no declaration is found for the variable there, the search continues in the declarations

of the subprogram that declared subprogram sub 1, which is called its **static parent.** If a declaration of *a* is not found there, the search continues to the next-larger enclosing unit and so on, until a declaration for *a* is found or the largest unit is declarations have been searched without success. In that case, an undeclared variable error has been detected. The static parent of subprogram sub 1, and its static parent, and so on up to and including the main program, are called the **static ancestors of sub 1.**

For example, consider the following Pascal procedure ;

```
procedure main ;
        var a : integer ;
        procedure sub 1 ;
        begin { sub 1 }
        ..... a .....
        end ; { sub 1 }
        procedure sub 2 ;
        var a : integer ;
        begin { sub 2 }
        .....
        end ; { sub 2 }
begin { main }
.....
end ; { main }
```

Fig 8.3 : A Skeleton Pascal Procedure.

Under static scoping, the reference to the variable *a* in sub 1 is to the a declared in the procedure main, because the search for a begins in the procedure in which the reference occurs, sub 1, but no declaration for *a* is found there. The search thus continues in the static parent of sub 1, main whose the declaration of *a* is found.

8.4.1.1 IMPORTANCE OF STATIC SCOPE

The importance of the static scoping is clearly highlighted in the following points.

1. The scope of a variable can be determined prior to execution (*i.e.* at compile time).
2. In the language that use static scoping, some variable declarations can be hidden from some subprograms.

3. The static type checking makes the program execution faster and mor reliable.
4. The static scoping makes program easier to read and understand.
5. Static scope rules play an important part in the design and implementatio of most programming languages *e.g.* Ada, C, FORTRAN, Pascal an COBOL.
6. The static scoping provides a method of accessing non-local variables.
7. The static scoping provides a different set of simplifications durin translation that make execution of the program more efficient.

8.4.2 DYNAMIC SCOPE

The scope of variable in APL, SNOBOL4, and the early versions of LISP i dynamic. The **dynamic scoping** is based on the calling sequence of subprograms not on their spatial relationship to each other. The **dynamic scope** of an association for an identifier is that set of subprogram activations in which the association i visible during execution. The dynamic scoping is thus named because the scope ca be determined only at runt time.

Dynamic Scope rule : A **dynamic scope rule** defines the dynamic scope o each association in terms of the dynamic course of program execution. The dynamic scope rules relate references with associations for names during program execution. We can define it according to the dynamic chain of subprogram activations.

For example, consider again the Pascal procedure shown in figure 8.3.

Under dynamic scoping rules, the meaning of an identifier a determined in sub 1 is dynamic *i.e.* it cannot be determined at compile time. It may reference the variable from either declaration of *a*, depending on the calling sequence.

The correct meaning of a can be determined at run time by beginning the search with the local declarations. When the search of local declarations fails, the declarations of the dynamic parent, or calling procedure are searched. If declaration for a is not found there, the search continues in that procedure's dynamic parent and so on, forming a **dynamic chain**, until a declaration for a i found. If none is found in any dynamic ancestor, it is a run-time error.

Consider the two different call sequences for sub 1 in the example above. First **main** calls **sub 2**, which calls **sub 1**. In this case, the search proceeds from the loca procedure, sub 1, to its caller, sub 2, where a declaration for a is found. So, reference to **a** in **sub 1** in this case is to the **a** declared in sub 2. Next, sub 1 is called

directly from **main**. In this case, the dynamic parent of **sub 1** is **main**, and the reference is to the **a** declared in **main**.

8.4.2.1 ADVANTAGES AND DISADVANTAGES OF DYNAMIC SCOPE

The dynamic scope provides following advantages.

1. There is no need to determine the scope at compile time.
2. The correct attributes of non-local variables can be easily determined at run time.
3. A statement in a subprogram can refer to different non-local variables during different executions of the subprogram.
4. In some cases, the parameters passed from one subprogram to another are simply variables that are defined in the caller. None of these need to be passed in a dynamically scoped language, because they are implicitly visible in the called subprogram.

However, the dynamic scope has following problems

1. There is no way to protect local variables from being accessed during the execution of the subprograms.
2. The dynamic scoping results in less reliable programs than static scoping.
3. The dynamic scoping is unable to statically type check references to non-locals.
4. Dynamic scoping makes programs much more difficult to read.
5. The accesses to non-local variables in dynamic scoped languages take for longer than access to non-locals when static scoping is used.
6. The implementation of dynamic scope rule is costly.
7. It is not possible to statically determine the declaration for a variable referenced as a non-local.

8.5 BLOCK STRUCTURE

The concept of block structured languages was originated in ALGOL 60, one of the most important early languages. Because of their elegance and effect on implementation efficiency, they have been adopted in other languages like Pascal, PL/I, and Ada etc.

The **Block structured languages** have a characteristic program structure and associated set of static scope rules. The block structured program is organized as a set of blocks. A **block** is a section of code having its own local variables whose scope is minimized. In general a **block** consist of :

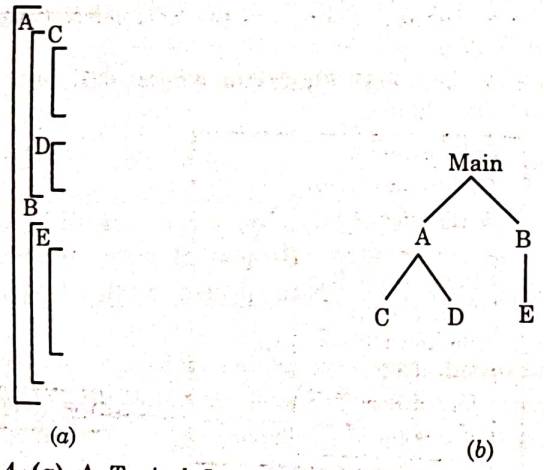
1. The beginning of a block consist of a set of declarations for names (or variables). Such variables are typically stack dynamic, so they have their storage allocated when the block is entered and deallocated when the block is exited. The declarations in a block define its local referencing environment.
2. The variable declarations are followed by a set of statements in which those names may be referenced.

The term **block-structure paradigm** is characterized by

(*a*) Nested blocks
(*b*) Procedures
(*c*) Recursion

These are described as given below :

(*a*) Nested blocks : In a block structured languages, the blocks may be nested within other blocks, and may contain their own variables. The nesting of blocks is accomplished by allowing the definition of one block to entirely contain the definition of other blocks. At the outermost level, a program consists of a single block, defining the main program. With in this block are other blocks defining subprograms callable from the main program. With in these blocks may be other blocks defining subprograms callable from which the first-level subprograms, and so on. Figure 8.4 (*a*) illustrates typical layout of a block-structured program. The corresponding tree representation of a block-structured program is shown in figure 8.4 (*b*).



(*a*) (*b*)

Fig. 8.4 (*a*) A Typical Layout of a Block Structured Program.
(*b*) The Tree Representation of Block Structured Program.

(*b*) Procedures : Once blocks have been implemented, procedures follow naturally as named blocks that can be called from other parts of a program, and that facilitate explicit information exchange between the calling and the called blocks through parameters.

(*c*) Recursion : The implementation model for blocks is the stack. Only one block can be active at any one time, and its allocated storage occupies the top of the run-time stack. When a block terminates, its memory allocation will be popped, and memory for the calling block reactivated. The stack implementation supports recursion, as successive invocations of a recursive procedure can be pushed onto the run-time stack and popped in reverse order, passing values back down the stack.

8.5.1 STATIC-SCOPE RULES FOR BLOCK STRUCTURED LANGUAGES

The static scope rules associated with a block-structured program are as follows :

1. The declarations at the beginning of a block defines the local referencing environment for the block.
2. If for an identifier referenced with in the body of block no local declarations exists, then the reference is considered as a reference to a declaration with in the block that immediately encloses the first block and so on.
3. Finally, if no declaration is found, the declaration in the predefined language environment is used, else the reference is taken as an error.
4. If a block contain another block definition, then any local declarations with in the inner block or any blocks it contains are completely hidden from the outer block and cannot be referenced from it *i.e.* inner blocks encapsulates declarations so that they are hidden from outer blocks.
5. The block name becomes the part of the local referencing environment of the containing block.

The static scope rules are clearly illustrated in figure 8.5.

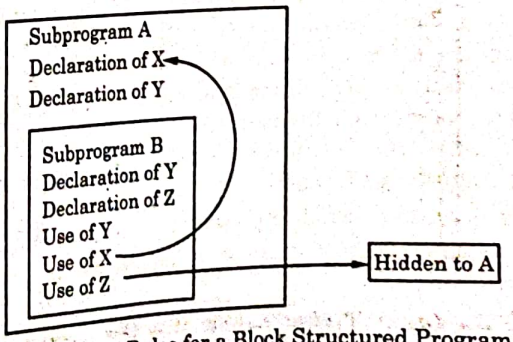


Fig. 8.5 Static Slope Rules for a Block Structured Program.

8.5.2 ADVANTAGES

The block structured language programs have following advantages.

1. A block introduces a new local referencing environment.
2. The program design is highly structured.
3. The variables declared in a block are stack dynamic.
4. Using the static scope rules, a declaration for the same identifier may occur in many different blocks ; but a declaration in an outer block is always hidden with in an inner block if the inner block gives a new declaration for the same identifier.
5. The static scope rules allow every reference to a name in a block to be associated with a unique declaration at compile time.
6. The compiler for the language may provide static-type checking.
7. Many simplifications of the run-time structure are based on the use of static scope rules.

8.6 LOCAL DATA AND LOCAL REFERENCING ENVIRONMENT

Subprograms are generally allowed to define their own data, called local data. The local data is defined inside the subprogram and access to the local data is usually restricted to the subprogram in which they are defined. Therefore the local **data** of subprogram defines the **local referencing environment**. The **local referencing environment** of a subprogram is the set of identifier associations available for use in referencing that may be determined without going outside the subprogram activation. The local data defined by the local referencing environment consist of

(*i*) **Local Variables**, declared in the head of the subprogram. These variables are called local because access to them is usually restricted to the subprogram in which they are defined. The local variables can be either static or stack dynamic. If the local variables are stack dynamic, they are bound to storage when the subprogram begins execution and unbound from storage when that execution terminates. The stack dynamic variables provides following advantages :

(*a*) Stack dynamic variables provides the subprogram flexibility.

(*b*) It is essential that recursive subprograms have stack-dynamic local variables.

(*c*) Some of the storage for local variables of all subprograms can be shared.

However the main disadvantages of stack-dynamic local variables are

(*a*) There is the cost of the time required to allocate, initialize and deallocates such variables for each activation.

(*b*) Access to stack-dynamic local variables must be indirect, whereas access to static variables can be direct.

(*c*) With stack-dynamic local variables, subprograms cannot be history sensitive *i.e.* they cannot retain data values of local variables between calls.

The static local variables have following advantages.

(*a*) The static local variables are very efficient.

(*b*) There is no indirection.

(*c*) They can be accessed much faster.

(*d*) They allow the subprograms to be history- sensitive.

The greatest disadvantage of static local variables is the inability to support recursion.

(*i*) Formal parameters (to be discussed in section 8.8)

(*ii*) The subprogram names defined locally with in that subprogram *i.e.* subprograms whose definitions are nested with in that subprogram.

8.6.1 SCOPE RULES FOR LOCAL REFERENCING ENVIRONMENT

For local environments, static and dynamic scope rules are easily made consistent :

Static scope rule : The static scope rule specifies that a reference to an identifier (or name) in the body of the subprogram is related to the local declaration for that identifier in the head of subprogram (assuming one exists). The static scope rules are implemented by means of a table of the local the entries are pairs, each containing an identifier and the associated data object. The name is only used so that later references to that variable will be able to determine where that variables will reside in memory during execution.

Dynamic scope rule : The dynamic scope rule specifies that a reference to an identifier (or name) during the execution of a subprogram refers to the association for that identifier in the current activation of the subprogram. Implementation of dynamic scope rule is done by following two methods.

(*i*) **Retention :** The identifier associations and the bound values are retained after execution. The implementation of the retention approach to local environment is straight forward using local environment tables. The table

is kept as the part of the code segment. Since the code segment is allocated storage statically and remains in existence throughout execution, any variables in the local environment part of the code segments are also retained. With this implementation of retention for the local environment, no special action is needed to retain the values of the data objects ; the values stored at the end of one call of subprogram will still be there when the next call begins. Also, no special action is needed to change from one local environment to another as one subprogram calls another. Since the code and local data for each subprogram are part of the same code segment, a transfer of control to the code for another subprogram automatically results in a transfer to the local environment for that subprogram as well. The languages like COBOL and many versions of FORTRAN use retention approach.

(ii) **Deletion :** In this approach, the identifier associations are deleted. The languages like C, Ada, LISP, APL and SNOBOL4 use the deletion approach. The implementation of the deletion approach to local approach to local environment is straight forward and using local environment tables. The table is kept as part of the activation record, destroyed after each execution. Assuming the activation record is created on a central stack on entry to subprogram, and deleted from the stack on exit, the deletion of the local environment follows automatically. Assuming each deleted local variable is declared at the start of the definition of subprogram, the compiler again can determine the number of variables and the size of each in the local environment table and may compute the offset of the start of each data object from the start of the activation record.

8.6.2 ADVANTAGES AND DISADVANTAGES

Both retention and deletion are used in a substantial number of important languages. These both approaches provides following advantages and disadvantages :

(i) The Retention approach allows the programmer to history sensitive subprograms.

(ii) The Retention approach consumes more storage space.

(iii) The Deletion approach is more natural strategy for recursive subprograms.

(iv) The Deletion approach provides a savings in storage space.

(v) The Deletion approach does not allow any local data to be carried over from one call to the next, so a variable that must be retained between calls must be declared as non-local to the subprogram.

8.7 SHARED DATA

The local data objects can be used only with in a single local referencing environment (*i.e.* with in a single subprogram). However, data objects are generally shared among several subprograms. The operations in each of subprograms may use the shared data. There are two ways that a subprogram can gain access to the shared data objects.

(i) Direct sharing through parameter transmission (to be discussed in section 8.8).

(ii) Through direct access to non-local referencing environments.

Although much of the required communication among subprograms can be accomplished through parameters, most languages provide direct access to non-local variables from external environments.

The **non-local variables** of a subprogram are those that are visible with in the subprogram but are not locally declared. **Global variables** are those that are visible in all program units. The global variables also form the part of non-local referencing environment.

There are four basic approaches to non-local environments that are used in programming languages :

(i) Explicit common environments and implicit non-local environments.

(ii) Dynamic scope

(iii) Static scope

(iv) Inheritance

These are four basic approaches are discussed as given below :

***(i)* Explicit common environment and implicit non-local environments:** The **explicit common environment** method is most straight forward method of sharing data. A set of data objects to be shared among a set of subprograms is located storage in a separate named block. Each subprogram contains a declaration that explicitly names the shared block. The data objects with in the block are then visible with in the subprogram and may be referenced by name in the usual way. Such a shared block forms the **common environment.**

Specification : A common environment is similar to a local environment, however it is not a part of any single subprogram.

- Definition of variables
- Constants, and
- types

A common environment cannot contain

- Subprograms
- Formal parameters

Implementation : The common environment is implemented as a separate block of memory storage. Special keywords are used to specify the variables to be shared. The data objects may then be referred by names in the usual way. The implicit non-local environment has already been discussed in section 8.3.3.

***(ii)* Dynamic Scope :** An alternative to the use of explicit common environments for shared data is the association of a non-local environment with each executing subprogram. A simpler, alternative is the use of the local environment for subprograms in the current dynamic chain.

Specification : The specification of dynamic scope has already been discussed in detail in section 8.4.2.

Implementation : An identifier association in the **dynamic chain** of subprogram calls is termed as **most-recent-association rule.** It is a referencing rule based on dynamic scope. The implementation of the most-recent-association rule for non-local references is straight forward, given the central-stack implementation for storing subprogram activation records. On entry to the subprogram, the activation record is created ; in return, the activation record is deleted.

***(iii)* Static Scope :** In block-structured languages like Pascal and Ada, the handling of non-local references to shared data is done by the static scope rules used during translation.

Specification : The specification of static scope has already been discussed in detail in section 8.41 and 8.5.1.

Implementation : The static scope rules are straight forward to implement in the compiler. The static scope rules can be implemented by using the following two techniques :

***(a)* Static Chain Implementation :** The static chain technique is the most direct technique for implementation of the correct referencing environment. A special entry in the beginning of the local environment table is called **static chain pointer.** This static chain pointer always contain the base address of another local table further down the stack. The table pointed to is the table representing the local environment of the statically enclosing block or subprogram in the original program. These static chain pointers forms the basis for a simple referencing scheme.

Advantages :

1. Fairly efficient implementation.
2. Allows straight forward entry and exit of subprograms.

Disadvantages : To determine an appropriate static chain pointer to install on ...try to a subprogram.

***(b)* Display Implementation :** The display implementation provides improvement upon the static chain implementation for non-local variable accessing. Instead of explicitly searching down the static chain for an identifier, we need only skip down the chain a fixed number of tables, and then use the base-address-plus-offset computation to pick out the appropriate entry in the table. We represent an identifier in the form of a pair (chain position, offset), during execution.

In this implementation, the current static chain is copied into a separate vector, ...rmed the **display** on entry to each subprogram. The display is separate from the ...ntral stack and is often represented in a set of high-speed registers. At any given ...int during execution, the display contains the same sequence of pointers that occur the static chain of the subprogram currently being executed.

Advantages :

1. Referencing using a display is particularly simple.
2. If the display is represented in high-speed registers during execution, then only one memory access per identifier reference is required.

Disadvantages : Subprogram entry and exit are more difficult.

***(iv)* Inheritance :** Often, the information is passed among program components ...plicitly. We call the passing of such information **inheritance.** The inheritance is ...e receiving in one program component of properties or characteristics of another ...mponent according to a special relationship that exists between the two ...mponents. We have often used inheritance in programming language design. The ...tails of further concept of inheritance are beyond the scope of this text.

...8 PARAMETERS AND PARAMETER TRANSMISSION SCHEMES

Subprograms typically describe computations. There are two ways that a ...bprogram can gain access to the data that it is to process.

(*i*) Through direct access to non-local variables (discussed in section 8.7).

(*ii*) Through parameter passing.

The data objects passed through parameters are accessed through names that ...e local to the subprogram. Parameter passing is more flexible than direct access to ...n-local variables. In essence, a subprogram with parameter access to the data it is ... process is a parameterized computation. It can perform its computation on ...hatever data it receives through its parameter.

8.8.1 PARAMETERS

An **argument** is the term used for a data object (or value) sent to a subprogram for processing. An argument can be obtained through :

(*i*) Parameters

(*ii*) Non-local references

The term **result** refers to the data object (or value) delivered by the subprogram returned through

(*i*) Parameters

(*ii*) Assignment to non-local variables

(*iii*) Explicit function values

Thus the terms **argument** and **result** apply to data spent to and returned from the subprogram through a variety of language mechanisms.

The parameters are associated with subprograms, specifying the form or pattern of data objects with which they will work. The parameters provide mechanisms for passing the information among subprograms. The different types of parameters are :

1. **Formal parameters :** A formal parameter is a particular kind of local data object with in a subprogram. The formal parameters are present in the subprogram header. It has a name and the declaration which specifies its attributes. They are sometimes called as dummy variables because they are not variables in the usual sense. In some case, they are bound to storage when the subprogram is called, and that binding is often through some other program variables. For example consider the following scenario as shown in figure 8.6.

```
void main ( )
        {
            int a, b ;
            .....
            .....
            sum (a, b) ;
            .....
            .....
        }
        sum (int x,  int y)
            {
            .....
            .....
            }
```

Fig 8.6 A C Program Fragment.

In figure 8.6 the subprogram **sum** defines two formal parameters x and y and declares the type of each.

2. **Actual parameters :** An actual parameter is a data object that is shared with the caller subprogram. The subprogram call statements must include the name of the subprogram and a list of parameters to be bound to the formal parameters of the subprogram. These parameters are called **actual parameters.** Actual parameters must be distinguished from the formal parameters because the two can have different restrictions on their forms, and of course their uses are quite different. An actual parameter may be :

(*i*) A local data object belonging to the caller,

(*ii*) A formal parameter of the caller.

(*iii*) A non-local data object visible to the caller.

(*iv*) A result returned by a function invoked by the caller and immediately transmitted to the called subprogram.

For example in figure 8.6 the subprogram defines two actual parameters a and b and declares the type of each.

8.8.2 ESTABLISHING THE CORRESPONDENCE BETWEEN PARAMETERS

In nearly all programming languages, the correspondence between actual and formal parameters or the binding of actual parameters to formal parameters needs to be established when a subprogram is called with a list of actual parameters. There are following two methods of establishing this correspondence.

1. **Positional Correspondence :** The simplest possible way to establish a correspondence between actual and formal parameters is by position. The actual and formal parameters are paired based on their respective positions in the actual and formal parameter lists *e.g.* the first actual parameter is bound to the first formal parameter and so forth. This is good method for relatively short parameter lists. Such parameters are called **positional parameters.**

2. **Correspondence by explicit name :** When the parameter lists are long, however, it is easy for the program writer to make mistakes in the order of parameters in the list. One solution is to provide **keyword parameters,** in which the name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter explicitly. The advantage of the keyword parameters is that they can appear in any order in the actual parameter list. For example in Ada

```
sum ( x ⇒ a,  y ⇒ b) ;
```

pairs the formal parameter x with an actual parameter a and a formal parameter y with an actual parameter b. The main disadvantage to keyword parameters is that the user of the subprogram must know the names of the formal parameters.

Some programming languages like Ada and FORTRAN 90 allow the keyword parameters to be mixed with the positional parameters in a call.

8.8.3 PARAMETER PASSING METHODS

The term **parameter passing** refers to the matching of actual parameters with formal parameters when a subprogram call occurs. The parameter – passing methods are the ways in which the parameters are transmitted to and/or from called subprograms. The two approaches are often used :

(*i*) Semantic models of parameter passing

(*ii*) Implementation models of parameter passing

These approaches are discussed in detail as given below.

8.8.3.1 SEMANTIC MODELS OF PARAMETER PASSING

To define the behaviour of the formal parameter, we use three distinct semantic models (called parameter modes).

(*i*) **IN mode :** An IN parameter mode lets you pass values to the subprogram being called *i.e.* formal parameters can receive data from the corresponding actual parameter. Inside the subprogram, an IN parameter acts like a constant. Therefore, it cannot be assigned a value. Figure 8.7 shows the IN mode of parameter passing.

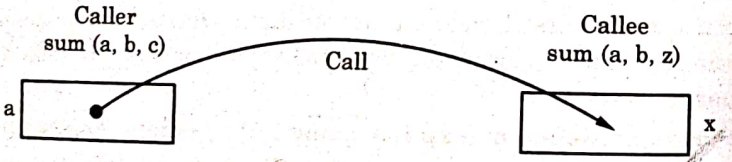


Fig. 8.7 An IN Mode of Parameter Passing

(*ii*) **OUT mode :** An OUT parameter mode lets you return values to the caller of a subprogram *i.e.* the formal parameters can transmit data to the actual parameter. Inside the subprogram, an OUT parameter acts like a variable. That means you can use an OUT formal parameter as if it were a local

variable. You can change its value or reference the value in any way. Figure 8.8 shows the OUT mode of parameter passing.

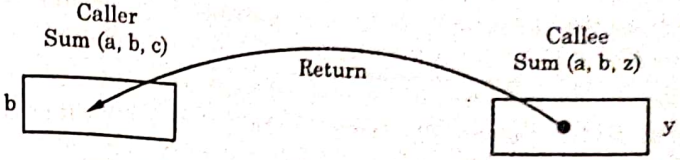


Fig. 8.8 An OUT Mode of Parameter Passing.

(*iii*) **IN OUT mode :** An IN OUT parameter mode lets you pass initial values to the subprogram being called and return updated values to the caller. Inside the subprogram, and IN OUT parameter acts like an initialized variable. Therefore, it can be assigned a value and its value can be assigned to another variable. he actual parameter that corresponds to an IN OUT formal parameter must be a variable ; it cannot be a constant or an expression.

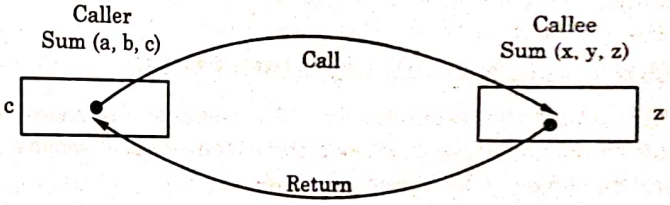


Fig. 8.9 An IN OUT Mode of Parameter Passing.

The table 8.2 summarizes the differences between three parameter modes described above.

IN mode	OUT mode	IN OUT mode
1. The default.	1. Must be specified.	1. Must be specified.
2. Passes values to a subprogram.	2. Returns values to caller.	2. Passes initial values to a subprogram and returns updated values to the caller.
3. Formal parameter acts like a constant.	3. Formal parameters act like a variable.	3. Formal parameter acts like an initialized variable.

IN mode	OUT mode	IN OUT mode
4. Formal parameter cannot be assigned a value.	4. Formal parameter must be assigned a value.	4. Formal parameter should be assigned a value.
5. Actual parameter can be a constant, initialized variable, literal, or expression.	5. Actual parameter must be a variable.	5. Actual parameter must be a variable.
6. Actual parameter is passed by reference (a pointer to the value is passed in)	6. Actual parameter is passed by value (a copy of the value is passed out) unless No copy is specified.	6. Actual parameter is passed by value (a copy of the value is passed in and out) unless No copy is specified.

Table 8.2 : Differences between IN mode, OUT mode and IN OUT mode

8.8.3.2 IMPLEMENTATION MODELS OF PARAMETER PASSING

Several models has been developed by language designers to guide the implementation of the three basic parameter transmission modes. The various methods (or modes) for transmitting parameters are

(*i*) Call by value
(*ii*) Call by result
(*iii*) call by value-result
(*iv*) Call by reference
(*v*) Call by name
(*vi*) call by constant value

These methods of parameter passing are discussed in detail as given below.

(*i*) Call by value : When a parameter is passed by value, the value of the actual parameter is used to initialize the corresponding formal parameter, which then act as a local variable in the subprogram, thus implementing IN-mode semantics. In call by value, the actual parameter is copied in the location of the formal parameter.

The call by value is normally implemented by actual data transfer because ocesses are usually more efficient with this method. In general a call-by-value plementation :

(*a*) Passes its *r*-value

(*b*) The formal parameter contains the value that is used.

The main disadvantage of the pass-by-value method if physical moves are done that the additional storage is required for the formal parameter, either in the lled subprogram or in some area outside both the caller and called subprogram. o, the changes made in the formal parameter values during execution of the bprogram are lost when the subprogram terminates.

As an example consider the following program skeleton shown in figure 8.10

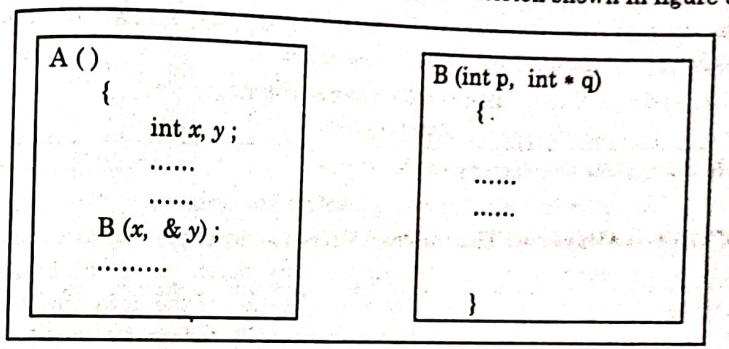


Fig. 8.10 A Program Skeleton with Calling and Called Subprogram

In figure 8.10, the subprogram B has two formal parameters *p* and *q*. The rameter *p* is transmitted by value here.

(*ii*) Call by result : The call-by-result is an implementation model for OUT mode parameters. A parameter transmitted by result is used only to transmit a result back from a subprogram. The formal parameter is a local variable with no initial value when the subprogram terminates, the final value of formal parameter is assigned as the new value of the actual parameter. For example in figure 8.10, the final value of *p* and *q* may be assigned as the new values of the actual parameters *x* and *y*. However the call by result transmission method has following disadvantages.

(*a*) It requires extra storage.

(*b*) The difficulty of implementing call-by-result by transmitting an access path usually result in it being implemented by data transfer.

(c) The problem of ensuring that the initial value of the actual parameter is not used in the called subprogram.
(d) There can be an actual parameter collision.
(e) The portability problems that are difficult to diagnose.
(f) A problem that an implement may be able to choose between two different times to evaluate the addresses of the actual parameters.

(iii) Call-by-value-result : The call-by-value-result is an implementation model for IN OUT mode parameters in which actual values are moved. It is in effect a combination of pass-by-value and pass-by-result. The value of the actual parameter is used to initialize the corresponding formal parameter which then acts as a local variable. In fact, call-by-value-result formal parameters must have local storage associated with the called subprogram. At subprogram termination, the value of the formal parameter is transmitted back to the actual parameter.

The call-by-value result has following disadvantages.

(a) Multiple storage for parameter
(b) Time for copying values
(c) The order in which actual parameters are assigned.

(iv) Call-by-reference : The call-by-reference is an implementation model for IN mode parameters. To transmit a data object as a call-by-reference parameter means that a pointer to the location of the data object is made available to the subprogram. The data object itself does not change position in memory. In the beginning of subprogram execution, the *l*-values of actual parameters are used to initialize local storage locations for the formal parameters. In general, a call-by-reference implementation.

(a) Passes its *l*-value.
(b) A call-by-reference parameter uses the *l*-value stored in the formal parameter to access the actual data object.
(c) There is no copying of values.

In figure 8.10, the parameter *q* is passed by reference.

(v) Call-by-name : The call-by-name is an IN OUT mode parameter transmission method that does not correspond to a single implementation model. When the parameters are passed by name, the actual parameter is, in effect, textually substituted for the corresponding formal parameter in all its occurrences in the subprogram. A call-by-name formal parameter is bound to an access method at the time of the subprogram call, but the actual binding to a value or an address is delayed until the formal parameter is assigned or referenced.

The call-by-name parameter passing method has advantages of the flexibility it affords the programmer. However, the call-by-name parameters passing method has following disadvantages.

(a) The process is slow relative to other parameter passing methods.
(b) High cost in terms of execution efficiency.
(c) Difficult to implement.
(d) Some simple operations are not possible with call-by-name parameters.

As an example of call-by-name parameter passing method in figure 8.10 the effect is to substitute the parameter *x* in place of *p* and *y* in place of *q*.

(vi) Call-by-constant value : If a parameter is transmitted by constant values then no change in the value of the formal parameter is allowed during program execution *i.e.* no assignment of a new value or other modification of the value of the parameter is allowed, and the formal parameter may not be transmitted to another subprogram except as a constant-value parameter. The formal parameter thus acts as a local constant during execution of the subprogram. The call-by-constant value has following advantages.

(a) Two implementations are possible.
(b) Protecting the calling program from changes in the actual parameter.
(c) The value of actual parameters cannot be modified by the subprogram.

8.3.3 IMPLEMENTATION OF PARAMETER TRANSMISSION

Each time when a subprogram is called a new activation of a subprogram receives a different set of parameters. Therefore, the storage for formal parameters is allocated as the part of activation record. Each formal parameter is a local data object in the subprogram. A formal parameter P can be treated as

(a) A Local data object of type T (*e.g.* in parameters transmitted by value-result, by value, and by result).
(b) A local data object of type pointer to T (*e.g* in parameters transmitted by reference).

The various actions associated with parameter transmission are split into two groups.

(a) Actions associated with the point of subprogram call in each calling subprogram.
(b) Actions associated with the entry and exit in the subprogram itself.

At each point of call, actual parameters are evaluated and the control is transferred to the called subprogram. After the control transfer, the **prologue** for the subprogram completes the actions associated with parameter transmission. Before the subprogram termination, the **epilogue** for the subprogram must copy the result values into the actual parameters transmitted by result or value result. The function values must also be copied into registers or into temporary storage provided by the calling program. The subprogram then terminates and activation record is lost, so all results must be copied out of the activation record before termination.

The compiler has two main tasks in the implementation of parameter transmission.

(a) It must generate the correct executable code for parameter transmission, for return of results and for each reference to a formal-parameter name.

(b) To perform the necessary static type checking to ensure that the type of each actual-parameter data object matches that declared for the corresponding formal parameter.

KEY POINTS TO REMEMBER

- The term **data control** is defined as the control of the transmission of data for each operation of a program.
- The two ways to make a data object available as an operand for an operation are direct transmission and referencing through a named object.
- A **name** is a string of characters used to identify some entity (or data object) in a program.
- An **association** is defined as the binding of identifiers (simple names) to particular data objects and subprograms.
- The **referencing environment** of the subprogram (or program) is defined as the set of identifier associations available for use in referencing during execution.
- The basic component of a referencing environment includes local referencing environment, Non local referencing environment, global referencing environment and predefined referencing environment.
- The referencing environment of a statement is the set of identifier associations (names) that are visible in the statement.
- The **scope** of a program variable is the range of the statements in which the variable is visible.

- A variable is **visible** in a statement if it can be referenced in that statement.
- The two types of scope rules are **static** and **dynamic** scope rules.
- The **static scope** of a declaration is the part of the program text where a use of the identifier is a reference to that particular declaration of the identifier.
- The **dynamic scoping** is based on the calling sequence of subprograms, and not on their spatial relationship to each other.
- The **block structured program** is organized as a set of blocks.
- A **block** is a section of code having its own local variables whose scope is minimized.
- The nesting of blocks is accomplished by allowing the definition of one block to entirely contain the definition of other blocks.
- **Subprograms** are generally allowed to define their own data, called local data.
- The local data of subprogram defines the local referencing environment.
- A subprogram can gain access to the shared data objects through :
  - — Direct sharing through parameter transmission.
  - — Through direct access to non-local referencing environments.
- An **argument** is the term used for a data object (or value) sent to a subprogram for processing and **result** refers to the data object (or value) delivered by the subprogram returned through parameters, assignment to non-local variable and explicit function values.
- The two different types of parameters are formal and actual parameters.
- The correspondence between parameters can be obtained by position or by explicit name.
- The three different parameter model are IN, OUT and INOUT modes.
- The various methods for parameter transmission are :
  - — Call by value
  - — Call by result
  - — Call by value-result
  - — Call by reference
  - — Call by name
  - — Call by constant value.

EXERICISE

1. Define the term data control. How it is different from sequence control ?
2. Write a short note on names and referencing environments.
3. What do you mean by a referencing environment ? Explain different components of a referencing environment.
4. Define the term scope. Explain static and dynamic scope.
5. What do you mean by a block structure ? Explain in detail.
6. What do you mean by a parameter ? Explain different parameter passing methods in detail.
7. Write short notes on :
   (a) Call by value and call by reference
   (b) Local data and Local referencing environment
   (c) Formal and actual parameters
   (d) Semantic models for parameter passing.

CHAPTER

9 STORAGE MANAGEMENT

9.1 INTRODUCTION

Memory or storage is where data and instructions are stored. The computer data storage refers to computer components, devices and recording media that retain digital data used for computing for some interval of time. For example, cache memory, main memory, floppy and hard disk are all storage devices. The main purposes of storage are described as given below :

(i) The computer data storage provides one of the core functions of modern computer, that of information retention.

(ii) Without significant amount of memory, a computer would merely be able to perform fixed operations and immediately output the result. It would have to be reconfigured to change its behaviour.

The storage management is the art and the process of coordinating and controlling the use of memory in a computer system. In its simpler forms, this involves providing ways to allocate portions of memory to programs at their request, and freeing it for reuse when no longer needed. The management of main memory is critical to the computer system. This chapter deals with various techniques for storage management.

9.2 GOALS OF STORAGE MANAGEMENT

Historically, a number of different memory management techniques have been used, and improved upon, in the operating system. The principle goals of the operating system's memory management are :

(i) To provide memory space to enable several processes to be executed at the same time.
(ii) To provide a satisfactory level of performance for the system users.
(iii) To protect each programs resources.
(iv) To share (if desired) memory space between processes.
(v) To make the addressing of memory space as transparent as possible for the programmer.

9.3 MAJOR RUN TIME ELEMENTS REQUIRING STORAGE

The major program and data elements requiring storage during program execution are as follows :

1. **Code segment for translated user programs,** regardless of whether programs are hardware or software interpreted.
2. **System run-time programs,** that support the execution of the user programs. These includes simple library routines, software interpreters or translators present during execution. It also includes the routines that control run-time storage management.
3. **User-defined data structures and constants,** declared in or created by user programs.
4. **Subprogram return points, coroutine resume points or event notices for scheduled subprograms.**
5. **Referencing environments (*i.e.* identifier associations),** as for example, the LISP A-list.
6. **Temporaries in expression evaluation,** *i.e.* temporary storage for the intermediate result of evaluation.
7. **Temporaries in parameter transmission** a list of actual parameters must be evaluated and the resuming stored in temporary storage until storage area where data are stored.
8. **Input-output buffers,** *i.e* a temporary storage area where data are stored between the time of the actual physical transfer of the data to or from external storage and the program – initiated I/O operations.
9. **Miscellaneous System data,** *e.g.* tables, status information for I/O, reference counts and garbage collection bits.
10. **Subprogram call and return operations,** *i.e.* storage allocation for subprogram activation record, the local referencing environment. However the subprogram return operations requires deallocation of storage.
11. **Data structure creation and destruction operations at arbitrary points during program execution,** *e.g* Pascal **new** and **dispose** operations and C **malloc** and **free** functions etc.

Component insertion and deletion operations, *e.g* the ML and LISP list operations that insert a component into a list.

9.4 PROGRAMMER AND SYSTEM CONTROLLED STORAGE MANAGEMENT

The term storage management has already been described in section 9.1. The storage management can be done.

(*i*) **Implicitly,** through the use of various languages features, called system controlled storage management.

(*ii*) **Explicitly,** by the programmer, called programmer controlled storage management.

The differences between programmer and system controlled storage management are summarized in table 9.1.

Programmer Controlled Storage Management	System Controlled Storage Management
1. In this, programmer explicitly has a control over storage *e.g.* C language provides this facility via malloc and free functions.	1. In this, storage is controlled by the system, implicitly itself. Many high level languages supports system controlled storage management.
2. It may place an extra burden over programmer.	2. In this, there is no burden over programmer.
3. Programmer controlled storage management is dangerous to programmer because it may lead to subtle errors or loss of access to available storage.	3. No such errors occurs in system controlled storage management.
4. Programmer may interfere with the necessary system controlled storage management.	4. No such interference of programmer is allowed.
5. The programmer knows quite precisely when a particular data structure is needed or when it is no longer needed and may be freed.	5. It is difficult for the system to determine when storage may be most effectively allocated and freed.

Table 9.1: Differences between programmer controlled and system controlled storage management

9.5 STORAGE MANAGEMENT PHASES

There are three basic storage management phases :

1. **Initial allocation :** Initially each piece of storage is either free or in use. If free, it is available for dynamic allocation as the execution proceeds. A storage management system requires some technique to keep track of free storage and mechanism for free storage allocation as the need arises during execution. The various run time elements that may require storage allocation have already been described in section 9.3. The storage allocation can be done by using any of the following technique.
   - *(i)* Static storage allocation
   - *(ii)* Stack based storage allocation
   - *(iii)* Heap based storage allocation
2. **Recovery :** The storage that has been allocated and is in use, must be recovered by the storage manager when the allocated storage becomes available for reuse. This involves finding the data objects which are no longer referenced (*i.e.* not required) and reclaim that memory. The return of newly freed storage to the free-space list is simple, provided such storage may be identified and recovered. The main problem lies in determining which elements are available for the reuse and therefore may be returned to the free-space list. The simplest recovery technique involves the explicit return by the programmer or system.
3. **Compaction and reuse :** The storage recovered may be immediately ready for reuse, or compaction may be necessary to construct large blocks of the free storage from small pieces. As the computation proceeds, the storage block is fragmented (divided) into smaller pieces through allocation, recovery and reuse. It is apparent that free-space blocks continue to split into ever smaller pieces. Ultimately one reaches a point where a storage allocator cannot honour a request for a sufficiently large size block request, even though, the free space list contains in total more that requested block. Therefore compaction is used to shuffle the memory contents to put all free memory together in one block. Depending upon whether active blocks with in the heap may be shifted in position, two approaches to compaction can be used.
   - *(i)* **Partial compaction :** If it is too expensive to shift active blocks (or active blocks cannot be shifted), then only adjacent free blocks on the free space list may be compacted.

   - *(ii)* **Full compaction :** If active blocks can be shifted, then all active blocks may be shifted to one end of the heap, leaving all free space at the other is contiguous block. Full compaction requires that when an active block is shifted, all pointers to that block be modified to point to the new location.

The reuse of storage after compaction involves the same mechanisms as initial allocation.

9.6 STATIC STORAGE MANAGEMENT

The simplest form of storage allocation is static storage allocation. In static storage allocation, objects are given an absolute address that is retained throughout the program's execution *e.g.* the storage allocation of global variables, non-recursive subroutine parameters etc.

The various elements in management are shown in figure 9.1.

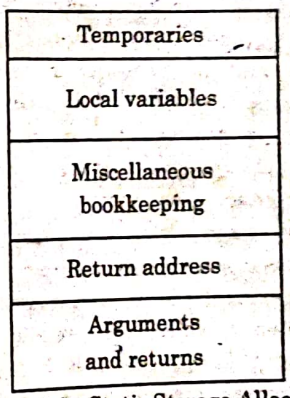


Fig. 9.1 The Static Storage Allocation

The space for all program and data can be allocated at compile time, *i.e.* statically if :

- *(i)* The size of every data item can be determined by the compiler *e.g.* no strings or arrays of adjustable length are permitted in the language.
- *(ii)* Recursive subprogram calls are not permitted.

This method of storage allocation is used in FORTRAN. Each subprogram can be compiled separately and the space needed for its arrays computed. Note that the only time a size need not be specified for an array occurs when that array is a formal

parameter. In that case, no storage need be allocated for the array at all. Since there is no recursion in FORTRAN, each subprogram can store its return address (place to which the return is to go) in a private location. Thus in FORTRAN, the space required for a program is just the sum of the space needed for the subprograms, their data and linkage information (return address) and any library routines used. This space never changes as the program is running.

A FORTRAN compiler can create a number of **data areas,** block of storage in which the values of names can be stored. In FORTRAN, there is one data area for each routine and one data area for each named common block and for blank common, if used. The symbol table must record for each name, the data area in which it belongs and its offset in that data area, that is, its position relative to the beginning of the area. The compiler must eventually decide where the data areas go, relative to the executable code and to one another.

For example, in FORTRAN, it is reasonable to place the data for each routine immediately after the code for that routine and to follow the routines by the common blocks. On some computer systems it is feasible to leave the relative positions of data areas unspecified and allow the link editor to link data areas and executable code.

The compiler must compute the size of each data area. For the routine's data areas, a single counter suffices, since their sizes are known after each routine is processed. For common blocks, a record for each block must be kept during the processing of all routines, since each routine using a block may have its own idea of how big the block is, and the actual size is maximum of the sizes implied by the various routines. If routines are separately compiled, the link editor must be able to select the size of a common block to be the maximum of all such blocks with the same name among the pieces of code being linked. The static storage allocation has the virtues that :

(*i*) It is easy to implement efficiently.

(*ii*) It requires no run-time support (library routines loaded with the object program) for allocating storage.

(*iii*) The translator can generate the direct *l*-value addresses for all data items.

(*iv*) Reduced freedom for the programmer.

(*v*) It eliminates the possibly of running out of memory.

(*vi*) Allows type checking during compilation

(*vii*) Compilation is easier

However, the main disadvantages of the static storage allocation are that

1. It is incompatible

(*a*) With recursive subprograms.

(*b*) With data structures whose size is dependent on computed or input data.

2. The size of the object must be known.

9.7 DYNAMIC STORAGE MANAGEMENT

If a programming language permits either recursive procedures or data structures whose size is adjustable, then some sort of dynamic management is necessary. The dynamic storage allocation provides memory space for a subprogram when it is called. When a subprogram is invoked, it needs space to store :

- Its parameters,
- Its local variables and
- The return address (the address in the calling code to which the computer returns when the subprogram completes its execution)

Such a space is called an **activation record.**

Consider a main program calls subprogram sub 1, which then calls subprogram sub 2. The diagram 1.2 shows how the successive activation records are created.

There are two kinds of dynamic storage allocation techniques.

(*i*) Stack Based storage management

(*ii*) Heap storage management

These two dynamic storage management techniques are discussed in the following subsections.

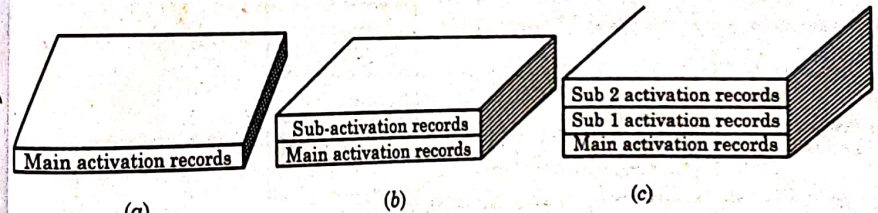


Fig. 9.2 Activation Records

(*a*) When the main program starts. (*b*) Main calls sub 1 (*c*) Sub 1 calls sub 2.

9.7.1 STACK BASED STORAGE MANAGEMENT

The simplest run-time storage management technique is the stack. The basic idea is to start with a region of consecutive words of memory. This region will be

used as a stack. The storage is allocated from the sequential locations in this stack block beginning at one end. Storage must be freed in the reverse order of allocation so that a block of storage being freed is always at the top of the stack.

A single stack pointer pointing to the stack top is needed to control the storage management. The allocated storage will always be added to the top by incrementing a stack pointer. Released storage will always be removed from the top by decrementing the stack pointer. While such a storage management scheme could not be used to handle arbitrary allocations and releases of storage, the last-in first-out mode of operation handles the basic storage requirements of a block-structured language.

The stack allocation is useful for handling recursive subprograms. As each subprogram is called, it places its data, on the top of a stack and when the subprogram returns, it pops its data off the stack. Virtually any language with recursive subprograms is required to use some sort of stack at run time. For example, in ALGOL, a name in a recursive procedure may have more than one activation at a given instant at run time. Therefore, dynamic allocation scheme involving a stack is usually used.

An example of stack-based allocation is shown in figure 9.3.

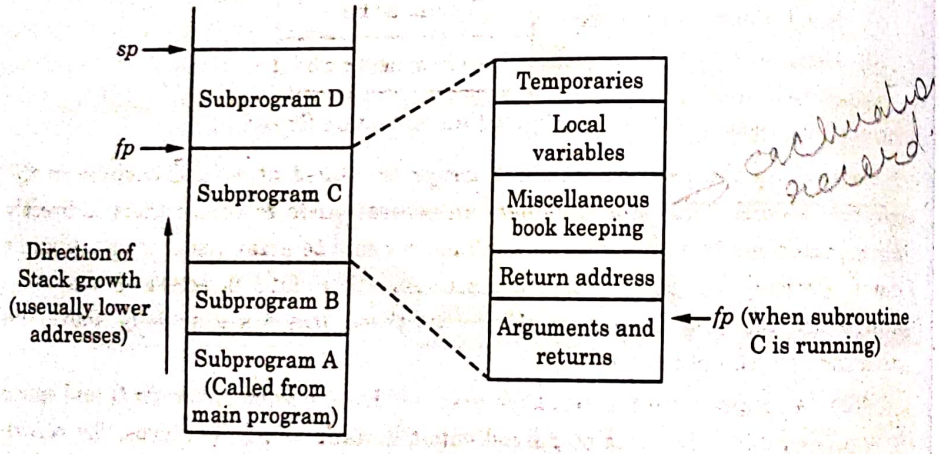


Fig. 9.3. A Stack-based Allocation Scheme.

We can collect all the fixed-size storage components declared in one subprogram into a single chunk of storage called an activation record. This activation record might contain the following items :

1. Storage for simple names, and pointers to arrays and other data structures local to the subprogram.
2. Temporaries for expression evaluation and parameter passing.
3. Information regarding attributes for local names and formal parameters, when these cannot be determined at compile time.
4. The return address.
5. A pointer to the activation record of the caller.

It is customary and useful to place immediately above the activation record data whose size cannot be determined until the subprogram is called. Figure 9.4 shows the format for a typical activation record, including two adjustable-length arrays.

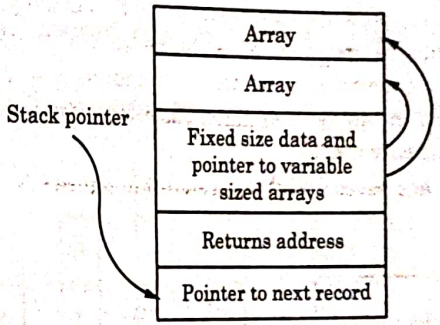


Fig. 9.4 An Activation Record

Pointers to the locations of these arrays are stored at a fixed position in the activation record. Thus, the compiler can generate code to access them indirectly through the top-of-stack pointer and the pointer for the array itself, even though it cannot precisely where the arrays begin. All other data is accessed indirectly, through the top-of-stack pointer. Note that these arrays are actually above the activation record, not part of it.

When a subprogram P calls subprogram Q, the activation record for Q and space for its adjustable –size data is pushed onto the stack. When Q returns, the return address is fetched from the activation record. Then the activation record for Q is effectively removed from the stack by lowering the stack pointer to the activation record for P immediately below.

As an example to illustrate the stack based storage management, consider a block-structured program as shown in figure 9.5.

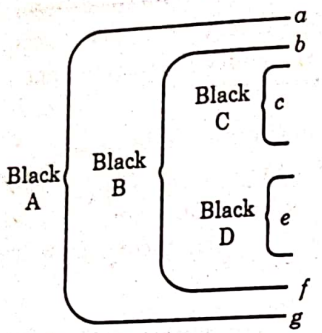


Fig. 9.5. A Block-Structured Program

The behaviour of stack at different moments is shown in figure 9.6.

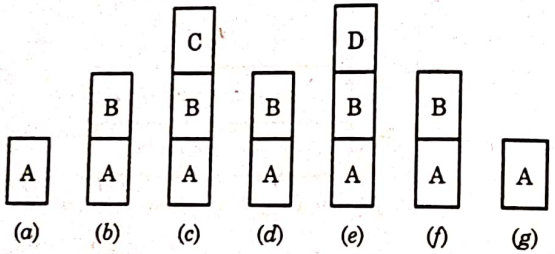


Fig. 9.6. Behaviour of Stack

When A begins execution, its activation record is placed on the top of the stack as shown in figure 9.6 (*a*). The stack contents when B and C initiate are shown in figure 9.6 (*b*) and 9.6 (*c*) respectively. The stack contents for points (*d*) through (*g*) of figure 9.5 are shown in figure 9.6 (*d*) through 9.6 (*g*), respectively.

In nutshell we can summarize the stack based storage managements as.

1. It provides clean and efficient support for nested subprograms and recursion.
2. The central concept is stack frame (also called activation record), that includes :

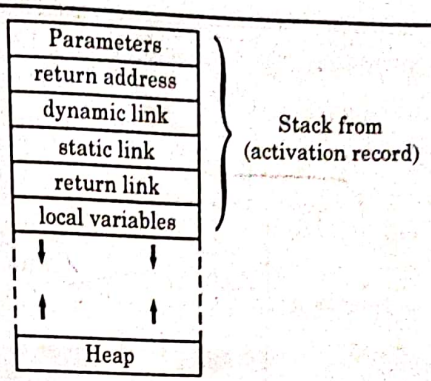


Fig. 9.7. A Stack Frame (Activation Record)

(*i*) Parameters
(*ii*) return address where to begin execution when the subprogram exits.
(*iii*) Dynamic-link-pointer to caller's stack frame.
(*iv*) Static-link-pointer to lexical parent (for nested subprograms).
(*v*) Return value-where to put the return value.
(*vi*) Local variables
(*vii*) Local work space – for temporary storage of results.

The order of activation follows the Last-In-First-Out rule *i.e.*

(*a*) Subprogram call – push stack frame
(*b*) Subprogram exit – pop stack frame

Each nested level of method invocation adds another activation record to the stack. As each method completes its execution, its activation record is popped from the stack.

The Depth of recursion is the number of activation records on the system stack, associated with a given recursive method.

The compaction occurs automatically as the part of freeing storage. Freeing a block of storage automatically recovers the freed storage and makes it available for reuse.

9.7.2 HEAP STORAGE MANAGEMENT

In the languages like LISP and SNOBOL, where data is constantly being created, destroyed and modified in size, it is inconvenient to place variable – length data on the stack. Also, a more complex strategy is required for languages a such as

PL/I, which allow the allocation and liberation of memory for some data in a non nested fashion.

One useful run-time organization is the heap, in which the storage can b allocated and freed arbitrarily from an area called a **heap**. The heap allocation i useful for implementing data whose size varies as the program is running, e.g., th strings in SNOLBOL or lists in LISP. It involves taking a large block of memory an dividing it into variable-length blocks, some used for data and some free. Thi arrangement is shown in figure 9.8.

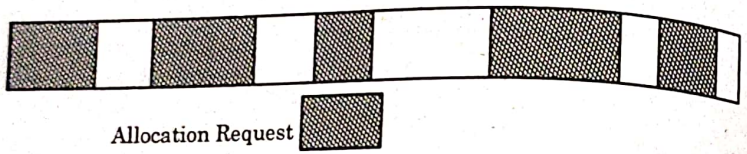


Fig. 9.8. A Heap Based Allocaton.

Figure 9.8 shows that the heap is a region of storage in which sub-block can b allocated and deallocated. In figure 9.8, the shaded blocks represent the allocate storage while the unshaded blocks represents free storage. The process of allocation and deallocation of storage is as

1. Initially the heap is maintained as a list of free space called free-space list The first word of the each item on the free list points to the first word of the next item on the free list as shown in figure 9.9.

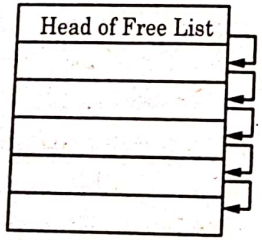


Fig. 9.9. Initial Free-space List

2. When a piece of data is created, the memory manager finds a free block of sufficient size to allocate the storage and marks this block as used and correspondingly changing the free space list (already shown in figure 9.8).
3. When a piece of data is no longer needed, the memory manager marks that block as free, the storage is deallocated and changing the free-space list.

Figure 9.10 shows a typical situation where we see the pointers from fixed locations representing three names X, Y and Z.

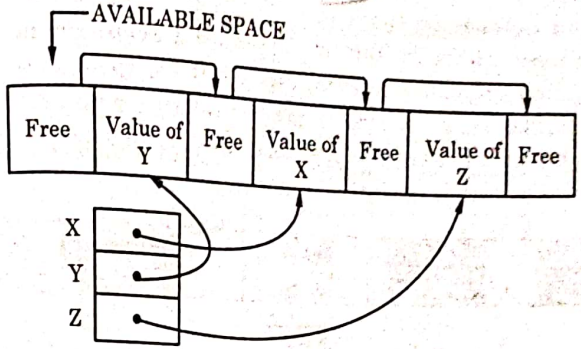


Fig. 9.10. Heap Allocation.

The value of each name, including a data descriptor giving the block's length, is kept in the block. The portions of the heap not currently in use are linked together in **available space list (free space list)**. That is, each free block contains a pointer to the next free block and information regarding how long the free block is.

In general, the heap storage management is required in the following situations.

1. The data whose size varies as the program is running.
2. The storage is required to be allocated and freed at arbitrary points during program execution *e.g.* objects created with C++ **new** and **delete.**
3. We can never know how big a data structure will become during one activation of the subprogram declaring it.
4. The storage allocation and deallocation is in a non-nested fashion *i.e.* recursion cannot be used.
5. In a typical programming language implementation, we will have both heap-allocated and stack-allocated memory that co-exit. In that case the available memory at the start of the program is divided into two areas : stack and heap. Both stack and heap share the same memory area, but grow towards each other from opposite ends as shown in figure 9.11.

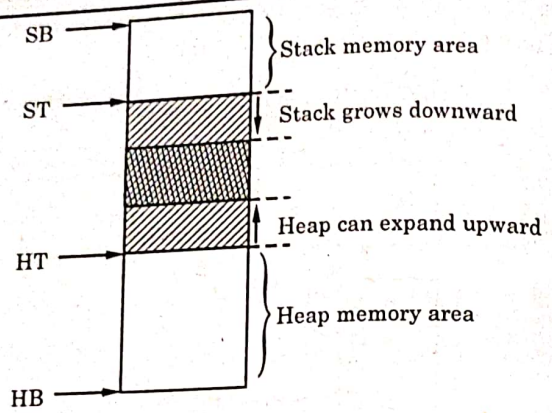


Fig. 9.11. Storage Allocation Where Stack and Heap Coexists.

It is convenient to divide heap storage management techniques into two categories, depending on whether the elements allocated are always of the same fixed size or of variable size. These are given as

(*i*) Heap storage management for fixed size elements.

(*ii*) Heap storage management for variable size elements.

These two different categories of heap storage are discussed in next two subsections respectively.

9.7.2.1. HEAP STORAGE MANAGEMENT FOR FIXED–SIZE ELEMENTS

When the elements allocated for storage management are always of fixed size, they are known as fixed size elements. For the fixed size elements, the storage management techniques are considerably simplified. Suppose that each fixed size element occupies N words of memory. Assuming the heap occupies a contiguous block of memory, we divide the heap block into a sequence of K elements, each N words long. Therefore, the size of heap is K × N. Whenever an element is needed, one of these is allocated from the heap. Whenever an element is freed, it must be one of these original heap elements.

Recovery : The return of newly freed storage to the free-space list is simple, provided such storage may be identified and recovered. The solution to identification are :

(*i*) Explicit return by programmer or system

(*ii*) Reference counts

(*iii*) Implicit return by garbage collector.

These techniques are explained below :

(*i*) **Explicit return by programmer or system :** The simplest recovery technique is that of explicit return. In explicit memory management, the program must explicitly call on operation to release memory back to the memory management system. For example table 9.2 shows the specific operations that must be called to free heap memory in Pascal, C and C++.

Language	Operation
Pascal	dispose
C	free
C++	delete

Table 9.2 : Specific operation to deallocate memory in different programming languages

where elements are used for system purposes (*e.g.* storage of referencing environments, return points, temporaries), each system routine is responsible for returning the space as it becomes available for reuse. However there are some problems with explicit return :

(*a*) **Garbage (Memory leaks) :** When all access paths to a data object are destroyed but the data object continues to exist, the data object is said to be **garbage**. The data object can no longer be accessed from other parts of the program, so it is of no further use, but the binding of data object to storage location has not been broken, so the storage is not available for reuse.

A data object that has become garbage ties up storage that might otherwise be reallocated for another purpose. A garbage element is available for reuse but not on the free-space list, and thus it has become inaccessible. Since the storage is finite, even in large mega by to machines, a build up of garbage can force the program to terminate prematurely because of insufficient memory in which to execute.

(*b*) **Dangling References :** A dangling reference is an access path that continues to exist after the life time of the associated data object. An access path ordinarily leads to the location of a data object *i.e.* to the beginning of the block of storage for the object. At the end of the life time of the object, this block of storage is recovered for reallocation at some later point to another data object. However, the recovery of the storage block does not necessarily destroy the existing access paths to the block, and this may continue to exist as dangling references.

The dangling references are a particularly serious problem for heap storage management, as they may compromise the integrity of the entire run-time structure during execution. For example, an assignment to a non-existent data object via a dangling reference can modify storage that has already been allocated to another data object of entirely different type, or it can modify housekeeping data that has been stored there temporarily by the storage management system thus destroying the integrity of the storage management system itself.

(ii) Reference counts : The use of reference counts requires explicit return, but provides a way of checking the number of pointers to a given element so that no dangling references are created. The various steps using this approach are :

(a) Associate a reference counter field, initially 0, with every allocated object.

(b) Each time a pointer is set to an object, up the reference counter by 1.

(c) Each time a pointer no longer points to an object, decrease the reference counter by 1.

(d) If reference counter ever is to 0, then free the object and return to the free-space list.

Using this approach :

(a) The garbage and dangling references can be avoided in most situations, the reference count is 0 only when nothing points to it.

(b) As all the objects are of fixed size, the fragmentation cannot occur.

(c) The inaccessible storage can still occur, but not easily.

The most important difficulty associated with reference counts is the cost of maintaining them.

(iii) Implicit return by garbage collector : This approach is mainly used in the languages like Java and Scheme. In implicit memory management, heap memory is reclaimed automatically by a **"garbage –collector"**. The main goal of the garbage collection is automatic reclamation of heap allocated-storage –application never has to free. In garbage collection, no attempt at keeping reference counts is made. Rather, when there is a path of pointers from the location associated with some name to that block. If not, the block is placed on the available free space list. This process is called **garbage collection.**

There are two statements to make you think about what a garbage collector really does.

1. A garbage collector provides the **"illusion of infinite memory"**.
2. A garbage collector predicts the future : it runs when **"almost"** about to run out of memory.

Now we discuss a two stage garbage collection algorithm.

We can view memory as a different graph where

- Each block is a node in the graph.
- Each pointer is an edge in the graph.
- Locations not in the heap that contain pointers into the heap are called **root** node (*e.g.* registers, locations on the stack, global variables). This concept is shown in figure 9.12.

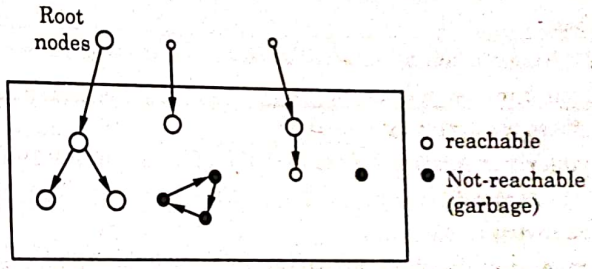


Fig. 9.12. Memory as a Directed Graph.

- A node (block) is **reachable,** if there is a path from any root to that node.
- The Non-reachable nodes are garbage (never needed by the application).

The two stages of garbage collection algorithm are

1. **Mark :** This stage :
   (a) Keep a linked-list (free list) of objects available to be allocated.
   (b) Allocate objects on demand.
   (c) Ignore free commands.
   (d) The active elements of heap (*i.e.* elements that are part of an accessible data structure) are marked by setting the garbage collection bit to **"off"** which was initially set **"on"**.

This means that mark stage starts at roots and set mark bit on all reachable memory.

2. **Sweep :** Once the marking algorithm has marked active elements, al those remaining whose garbage collection bit is "on" are garbage and ma be returned to the free-space list. *i.e.* sweep stage scan all blocks and fre blocks that are not marked.

The mark and sweep algorithms are given in figure 9.13.

```
ptr mark (ptr p) {
    if (! is _ptr (p)) return ;          // do nothing if not pointer
    if (mark Bit set (p)) return         // check if already marked
    setmarkBit(p) ;  // set the mark bit
    For (i = 0 ; i < length (p) ; + +)   // mark all children
    retun ;
}
```

(*a*)

```
ptr sweep (ptr p, ptr end) {
while (p < end) {
    if markBitset(p)
      ClearMarkBit ( ) ;
    else if (allocate Bit set (p))
      free (p) ;
      p + = length (p) ;
}
```

(*b*)

Fig 9.13 : (*a*) : Mark using depth-first transversal of the memory graph
(*b*) : Sweep using lengths to find next block

The algorithm given in figure 9.13 have the following assumptions.

Application :

1. new (n) : returns pointer to new block with all locations cleared.
2. read (b, i) : read location i of block b into register.
3. write (b, i, v) : write v into location i of block b.

Each block will have a header word

1. Addressed as b [– 1], for a block b.
2. Used for different purposes in different collectors.

Instructions used by the Garbage Collector

1. is_ptr (p) : determines whether p is a pointer.
2. length (b) : returns the length of block b, not including the header.
3. get_roots ( ) : return all the roots.

An example using mark-sweep algorithm is shown in figure 9.14.

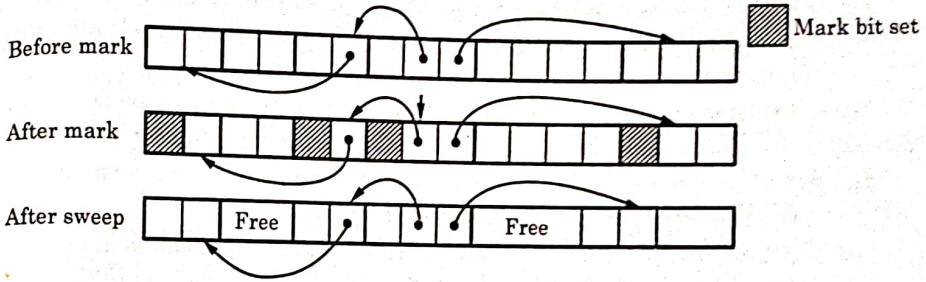


Fig. 9.14. An Example of **Mark-Sweep** Garbage Collection Algorithm.

There are most basic issues that needs to be discussed regarding garbage collection.

***(i)* How does the memory manager know when memory can be freed ?**

In general, we cannot know what is going to be used in the future since it depends on conditionals. But we can tell that certain blocks cannot be used if there are no pointers to them.

***(ii)* Need to make certain assumptions about pointers**

- Memory manager can distinguish pointers from non-pointers.
- All pointers point to the start of a block.
- Cannot hide pointers (*e.g.*, by coercing them to an int, and then back again)

Note : The compaction is not a problem with fixed-sized elements as all elements have same size.

9.7.2.2 HEAP STORAGE MANAGEMENT FOR VARIABLE – SIZE ELEMENTS

When the elements allocated for storage management are of varying size (*i.e.* the different elements for which storage is to be allocated are of different sizes), they are known as **variable-size elements**. For the variable size elements, the storage

management techniques are more difficult than with fixed – size elements, although many of the same concepts apply. The examples where variable – size elements occur include :

(i) Programmer – defined data structures such as arrays.

(ii) Activation records for tasks.

Initial Allocation and Reuse : The basic steps for initial allocation of storage are :

(i) The heap is considered as a large block of free storage. A heap pointer is appropriate for initial allocation.

(ii) When a block of N words is requested, the heap pointer is advanced by N and original heap pointer value returned as a pointer to the newly allocated element.

When the heap pointer reaches the end of the heap block, some of the free space ack in the heap must be reused. There are following two possibilities for reuse.

(i) Search the free-space list for an appropriate – size block and return any left over space to the free – space list after the allocation.

(ii) Compact the free space so that all the active elements are moved at one end leaving the free space at the other end.

(i) Reuse from a free-space list : When a request for an N-word element arises, the free-space list is scanned for a block of N or more words. A block of N words can be allocated dynamically. A block of more than N words is usually splitted into two blocks, an N-word block which is allocated and the remainder block, returned to the free space list. The following two techniques are used for managing the allocation from a free-space list. There are as given below.

(a) First-Fit Method : On a request of N-word block, the free-space list is searched for the **first** block of N or more words. If a block of more than N words is found, it is splitted into two parts one of size N to be allocated and the remainder block is returned the free-space list.

(b) Best-fit Method : On a request of N-word block, the free-space list is searched for a block of **minimum** number of words greater than or equal to N. If the block of size N, it is allocated completely, else the block is splitted into two parts : first consisting of size N which is allocated immediately and the remainder is returned to the free-space list.

(ii) Recovery with variable-size blocks using compaction : First of all we examine the three methods of recovery used for fixed sized elements in context with variable size elements.

(a) Explicit return is the simplest possible technique, but the problems of garbage and dangling references are again present.

(b) Reference counts can be used in the ordinary manner.

(c) Implicit return by garbage collection is also a feasible technique.

The main difficulty now is in the collecting phase of mark-sweep algorithm. In addition to the garbage – collection bit in the first – word of each block (active or not), an integer **length – indicator** specifying the length of the block is also maintained. This makes, a sequential scan of memory possible, looking only at the first word of each block.

The main problem that any heap storage management system using variable – size elements faces is that of **memory fragmentation**. In fragmentation ;

- Blocks are split to create smaller blocks to fit requested size through allocation, recovery and reuse.
- Blocks are never merged.
- Blocks will get smaller and smaller and will run out of memory without really being out.

(i) Internal Fragmentation :

- If size of object to allocated is larger than the size of the available heap.
- Come from fixed size heap blocks.

(ii) External Fragmentation :

- If size of object to allocated is not larger than the size of the available heap, but the available space in the heap is scattered through heap in such a way that no contiguous space fits.
- Comes from variable size blocks.

To fight fragmentation using memory performs **"heap compaction"** once in a while. The process of compaction refers to construct large blocks of the free storage from the small blocks by merging them together. The general concept of the compaction is shown in figure 9.15.

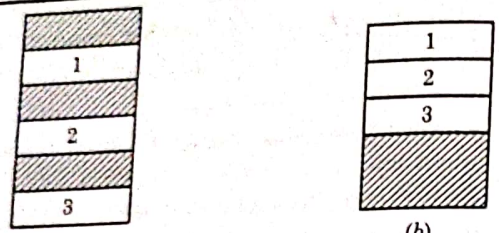


Fig. 9.15 (*a*) Before Compaction (*b*) After Compaction

Depending on whether active blocks with in the heap may be shifted in position, two approaches to compaction can be used.

(i) **Partial Compaction :** If it is too expensive to shift active blocks or the active blocks cannot be shifted, then only the adjacent free blocks on the free space list may be compacted.

(ii) **Full Compaction :** If the active blocks can be shifted, then all active blocks may be shifted to one end of the heap, leaving all free space at the other in contiguous block. Full compaction requires that when an active block is shifted, all pointers to that block be modified to point to the new location.

The reuse of storage after compaction involves the same mechanisms as initial allocation.

KEY POINTS TO REMEMBER

- The **memory** or **storage** is where data and instructions are stored.
- The **storage management** is the art and process of coordinating and controlling the use of memory in a computer system.
- The major run time elements requiring storage are :
  - Code segments for translated user programs
  - System run-time programs
  - Use defined data structures and constants
  - Subprogram return points, coroutine resume points or event notices for scheduled subprograms.
  - Referencing environments
  - Temporaries in expression evaluation
  - Temporaries in parameter transmission
  - Input-Output buffers
  - Miscellaneous system data
  - Subprogram call and return operations
  - Data structure creation and destruction operations
  - Component insertion and deletion operations.
- The storage management can be either **programmer controlled** or **system controlled**.
- The three basic storage management phases are :
  - Initial allocation
  - Recovery
  - **Compaction and Raves**
- The two basic storage management techniques are
  - **Static storage management**
  - **Dynamic storage management**
- In **static storage management**, the storage is allocated at compile time and recursive subprogram calls are not permitted.
- In **dynamic storage management**, the storage is allocated dynamically during execution and recursive calls are permitted.
- The dynamics storage management can be either
  - **Stack based storage management**
  - **Heap storage management**
- The stack based storage management uses LIFO stack.

EXERICISE

1. Define the term storage and storage management. What are different goals of storage management ?
2. What are major run time elements requiring storage ?
3. Explain the storage management phases in detail.
4. Explain the static and dynamic storage management in detail.
5. What is stack based storage management ? Explain in details.
6. Define the term heap storage management. Explain the concept of heap storage management for fixed size and variable size elements in detail.
7. Write short notes on :
   (*a*) Programmer and system controlled storage management
   (*b*) Fixed size and variable size elements
   (*c*) Garbage and dangling references
   (*d*) Mark-sweep algorithm.

CHAPTER

10 PROGRAMMING LANGUAGES

10.1 INTRODUCTION

A programming language is a notation for writing computer programs used for specifying, organizing and reasoning about computations. The programming languages are used to facilitate the communication about the task of organizing and manipulating information and to express algorithm precisely. The complete description of a programming language includes the computational model (*i.e.* a collection of values and operations), the syntax, semantics and pragmatic considerations of programs that shape the language. The present chapter deals with different programming language paradigms.

10.2 PROGRAMMING LANGUAGE PARADIGMS

During the last 10 years, languages for programming computers have been organized into a hierarchy of paradigms, the major ones being those shown in figure 10.1.

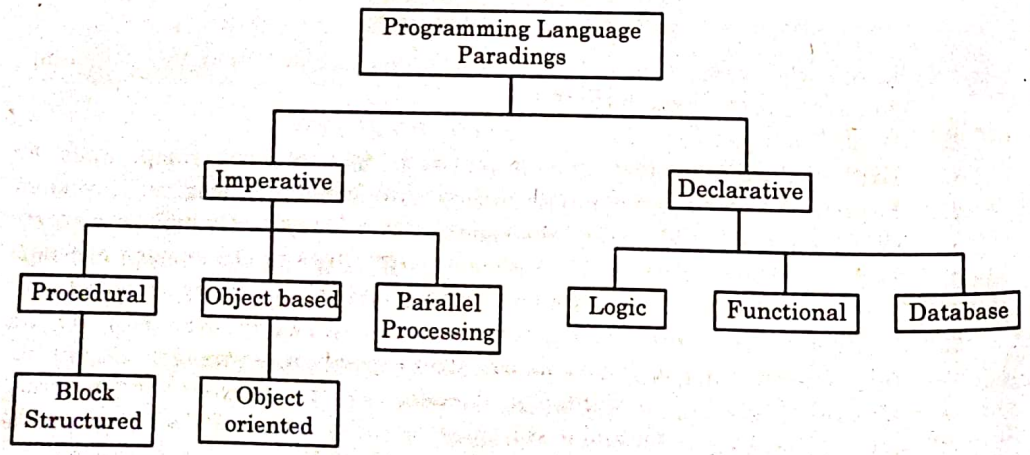


Fig. 10.1. A Hierarchy of Porgramming Language Paradigms.

A paradigm can be thought of as a collection of abstract features that categorize a group of languages which are accepted and used by a group of practitioners. The notion of scientific paradigms can be found in Thomas Kuhn's **The structure of Scientific Revolutions.** He defines them as "universally recognized scientific achievements that for a time provide model problems and solutions to a community of practitioners". Peter Wegner extends the notion to programming language paradigms, which "may be defined intensionally by their properties or extensionally by one or more instances".

In defining a paradigm, no attempt is made to insure that the list of language exemplars is exhaustive. Kuhn uses the term exemplar for an example that helps to active. A single language incorporating all the features of a paradigm is thus an exemplary realization of the paradigm. We may investigate a particular paradigm by exploring the features of one or more representative languages. You will not be wrong if you relate a language paradigm to a very good example language for a collection of related ideas.

A paradigm, and also its exemplars, is most useful when it is simple and clearly differentiates one language from another. Exemplars may be manufactured to serve as a model, as are some experimental languages, or may be already existing languages. We may say that Ada is **"block – structured"** and also **"object – based"**. Thus Ada belongs to both the block structured and object – based paradigms. Whether or not Ada can serve as an exemplar depends on your view of Ada.

As shown in figure 10.1, the paradigms for programming languages fall into two classifications,

(*a*) Imperative, and

(*b*) Declarative

These are described as given below.

(*a*) Imperative Languages :

In **Imperative paradigms** are those which facilitate the computation by means of state changes. By a state, we mean the condition of a computer's random access memory (RAM), or store. It is sometimes helpful to think of computer memory as a sequence of "snapshots", each one capturing the values in all memory cells at a particular time. Each individual snapshot records a state.

When a program is entered, associated data exists in a certain condition, say an unsorted list off-line. It is the programmer's job to specify a sequence of changes to the store that will produce the desired final state, perhaps a sorted list. The store involves much more than data and a stored program, of course. It includes symbol tables, a run-time stack (S), an operating system and its associated queues and stacks etc. The complete program, data, and even the CPU itself can be viewed as part of the initial state.

In general, the Imperative languages are action oriented ; *i.e.* a computation is viewed as a sequence of actions.

We can think of the imperative languages collectively as a progression of developments to improve the basic model, which was **FORTRAN 1**. All have been designed to make efficient use of **Von–Neumann architecture computers.** The **Backus** anticipated the two key ingredients of FORTRAN'S success ; familiar notations and efficiency.

Efficiency was emphasized because it was very on the 1950s that translation would be inefficient. That is, hand – crafted machine language programs would be shorter and would run faster than the results of translation. The examples of Imperative programming languages includes ALGOL, PASCAL, C, C++ etc. The various paradigms under the imperative languages are as discussed below.

(i) Procedural Languages :

Procedural languages are command driven or statement oriented language. A program consists of a sequence of statements, and the execution of each statement causes the interpreter to change the value of one or more location in its memory that is enter a new state. Syntax of such languages generally have the format.

statement 1 ;
statement 2 ;

C, Pascal, FORTRAN and similar languages are procedural languages *i.e.*, each statement in the language tells the computer to do something. Get some input, add these numbers, divide by 6, display the output. A program in a procedural language is a list of instructions. For very small programs, no other organising principle is needed. Such language consists of series of procedures (or subprogram or functions or subroutines) that execute when called. each procedure consists of a sequence of statements where each statement manipulates data that may be either local to the procedures, a parameter if from the calling procedure or defined globally. The procedural Paradigm's shown in figure 10.2.

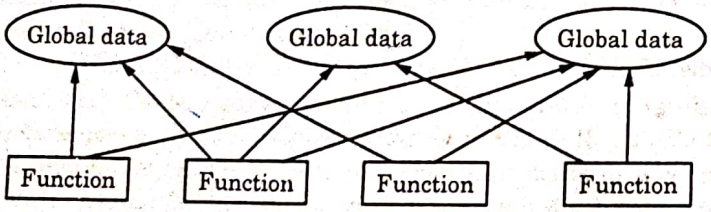


Fig 10.2. The Procedural Paradigm

Local data for each activation record associated with that procedure and data stored in such activation records typically have relatively simple types such as integers, real, character and boolean. e.g. FORTRAN and C FORTRAN uses static storage allocation while C is characterized by dynamic activation record creation.

In Procedural languages, it is very difficult to define user-defined data types. It only supports built-in data types. It has limited set of control structures of data is not supported well.

(ii) Object Oriented Programming Languages

In dividing programming language into two paradigms : **imperative and declarative**, each with three sub-paradigms, object-oriented languages have been placed in the imperative paradigm, since it was in the imperative language simula that these notions began.

The fundamental idea behind object oriented languages into a single unit both data and the functions that operates on the data. Such a unit is called an object. It not only primarily concerned with the details of program operations. Instead, it deals with the overall organisation of the program. OOPs enables the user to remain close to the conceptual, higher level model of the real world problem that is to be solved, especially in large software projects. The user can manage the complexity better than with approaches that force you to map the problem to fit the features of the language. The object oriented Paradigm is shown in figure 10.3.

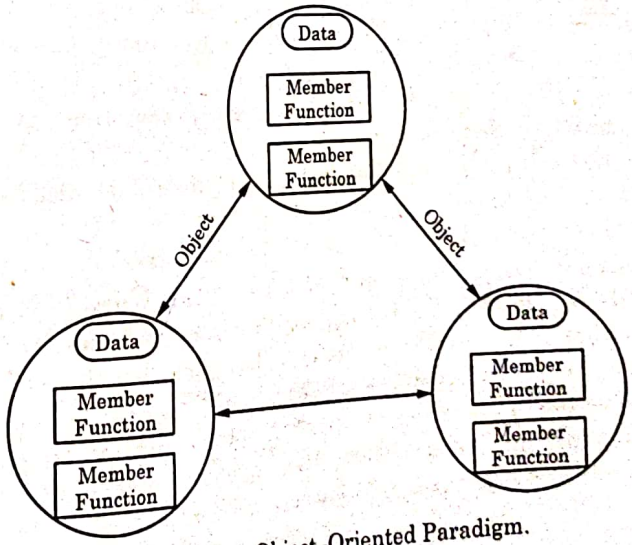


Fig. 10.3. The Object–Oriented Paradigm.

The object-based paradigm describes languages that support interacting objects. An object is a group of procedures that share a state. Since data is also part of a state, data and all the procedures or functions that apply to it can be captured in a single object.

An object is defined as **"a group of procedures that shares a state"**. A program written in an imperative language involves in transitions commands. An object is an item or thing with its associated, well – defined behaviours.

We will define an object as a collection of data, called its states, and the procedures capable of altering that state. If an object is a simple robot consisting of a movable arm and a gripper, its state will include its position in the room where it is located, the angle of the arm and whether its gripper is open or closed. A robot object must have a name, to distinguish it from other robots.

The objects are called packages ; Modula. where they are called modules ; and small talk, where objects are called (right fully) objects.

In C++, a collection of objects is grouped into class.

The term object oriented was originally used to distinguish those object-based languages that supported classes of objects and inheritance of attributes of attributes of a parent object by its children, object oriented languages are much more complex than structural and procedural languages.

The mapping between functions and data is done using the concept of **'object'**. In an object data is binded to a function. The major change is the addition of classes and a mechanism for inheriting class objects into other classes. The design of C++ was guided by three principles :–

(*i*) The use of classes would not result in programs executing any more slowly than programs not using classes.

(*ii*) In order to make sure that the addition of classes did not cause needed features to be omitted.

(*iii*) No run time inefficiency should be added to the language.

It easily supports both primitive and user defined data-types. In most cases the association of a method and the source program that must be executed is determined statically at translation time. Object oriented language has the same sequence control statements as structural language except an added exception handling.

It also supports overloading of functions. In object oriented language call by value mechanism is used for parameter passing and pointers may be used to implement call by reference. *e.g.* C++, Java etc.

(*iii*) Parallel Processing Languages :

When there is a single execution sequence, the program is termed as a sequential program because execution of its subprograms proceeds in a predefined sequence. In the more general case, where several subprograms might be executing simultaneously, the program is termed as a concurrent or parallel program. Each subprogram can execute concurrently with other subprograms.

The term **parallel** implies the positions of multiple processors, while **concurrent** suggests that processors are running at the same time. The terms are often interchangeable. The following three issues distinguish concurrent from sequential programming :

(*i*) The use of multiple processors.

(*ii*) Cooperation among the processors.

(*iii*) The potential for partial failure *i.e.*, one or more processes may fail without jeopardizing the entire project.

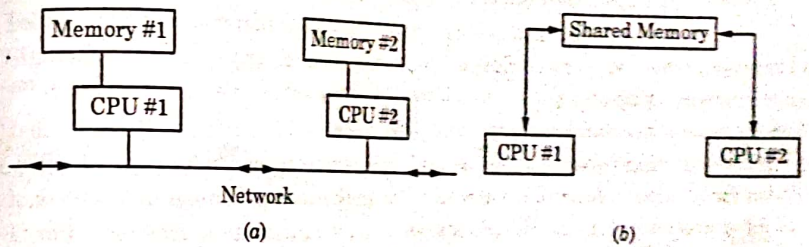


Fig. 10.4 The Physical Models for Parallel Processing.

The Labels have been interconnecting is no unanimity in the literature for naming what each of the two diagrams represent. All agree, however, that both involve two or more CPUs that communicate with each other. The top diagram in figure 10.4 shows each with its own memory and a communication channel between them. Here memory as well as CPU are distributed. The bottom diagram in figure 10.4 shows no communication channel, but shared memory. Only the CPU are distributed.

The examples of parallel processing languages includes, **Ada, Pascal S, Occam, C-Linda etc.**

(*b*) Declarative Programming Languages :

Unlike an imperative languages, which allows one to write a sequence of commands to a computer, a declarative language facilitates the writing of declarations, or truths. The Declarative languages are considered to be at higher

level than imperative languages, because a declarative programmer works with concepts rather than with storage locations of the machine itself.

A declarative language is one in which a program specifies a relation or function, in which the programmer does not consider the assignment of values to storage locations, but thinks in terms of functional values or the relationship of entities to each other when solving a problem. The interpreter or compiler for the particular language manages memory for us. The following three declarative paradigms are taken from mathematics :

(*i*) Logic

(*ii*) The Theory of functions, and

(*iii*) The Relational calculus

The examples of declarative programming languages includes **PROLOG, LISPET**. The various languages paradigms under the declarative programming languages paradigm are as discussed below.

(i) The Logic Programming Languages : The programming that uses a form of symbolic logic (predicate calculus) as a programming language is often called **logic programming** and languages based on symbolic logic are called **logic programming languages.**

Logic is the science of reasoning, and as such includes formal methodologies that have been found useful for solving problems other than those resolved by intuition, leaps of faith or compromise. In a logic programming language, a program consists of a set of statements (written informs known as Horn clauses) that describe what is true about a desired result, as opposed to giving a particular sequence of statements that must be executed in a fixed order to produce the result. The predicate calculus provides axioms and rules so one can deduce new facts from other known facts. A Horn clause allows only one new fact to be deduced in any single statement. A system of Horn clauses allows a particularly mechanical method of proof called resolution.

A logic-based program consists of a series of axioms or facts, rules of interference, and a theorem or query to be proved. The output is true if the facts support the query and false otherwise.

A pure logic programming language has no need for control abstractions such as loops or selection. Control is supplied by the underlying system. All that is needed in a logic program is the statement of the properties of the computation. For this reason, logic programming is sometimes called **declarative programming,** since properties are declared, but no execution sequence is specified. The example of a

logic programming language includes PROLOG. The syntax of such languages is similar to :

enabling condition 1 → action 1
enabling condition 2 → action 2
⋮
enabling condition n → action n

This means that if enabling condition : is true then corresponding action performed is action i.

PROLOG

The language **PROLOG** was implemented (in Fortran) by Alain Colmerauer in the early 1970s. Prolog is the predominant language in artificial intelligence. The name Prolog is short for programming Logic. It is one of the only programming languages based on pure logic. It has an internal set of specifications in formal logic. Its background theories are underlined with first order predicate calculus. First order predicate calculus gives Prolog very clean and elegant syntax along with well-defined semantics.

Prolog is a **Rule-Based Programming Language.** Basically, this means that the programmer types in a set of rules for a condition to be true. In order for the condition to be true, the computer must check to make sure all the rules are complied. If for any reason a rule is not met the condition fails.

One of the best features of Prolog is that it is a declarative language. This allows the programmer to construct the program in terms of problem's constraints, and not performing an algorithm. To the computer, this means the program tells it "what is true" and "what needs to be done" instead of "how to do it".

Characteristics of Logical Languages :

1. Logical programming is declarative ; specify what do compute, not how
2. Computation consists of logical deduction
3. Execution apply rules to facts and generate new facts
4. Backtracking search is built into the execution mechanism
5. Logical programming is based on a logical inference called resolution
6. Resolution : unify a query with a fact or the head of a rule

(ii) Functional Programming Languages : An alternative view of the computation performed by a programming language is to look at the function that the program represents, rather than just to state changes as the program executes, statement by statement. We can achieve this by looking at the desired result rather than at available data. The languages that emphasize this view are called applicative or functional languages.

The functional paradigm bases the description of computation on the evaluation of functions or the application of functions to known values. For this reason, functional languages are sometimes called **applicative languages.** The basic objects of functional languages are atoms and list of atoms with function call and recursion being the basic execution mechanism. This involves, besides the actual evaluation of functions, the passing of values as parameters to functions and the obtaining of the resultant values as returned values from function. The functional paradigm involves no notion of variables or assignment to variables. The syntax of such languages is similar to

function n (.....(function 2 (function, (data)) ...)

The examples of functional languages includes LISP and ML etc. In general, the functional languages differs from other languages in following three respects :

(*i*) From their design they are meant to be programmed applicatively.

(*ii*) The execution of expression is the major concept needed for programming sequencing rather than the statement of languages like C, Pascal or Ada.

(*iii*) Since they are applicative (or functional) heap storage and list structures become the natural process for storage management rather than the traditional activation record mechanism in procedural languages.

The function paradigm provides the following advantages :

(*i*) The language becomes more independent of the machine model and that, because the functional programs resemble mathematics.

(*ii*) It is easier to draw precise conclusions about their behaviour.

LISP

Developed in the late 1950s by John McCarthy at MIT, **LISP** is considered the grandfather of all artificial intelligence languages. The name stands for **List Processing languages.** It contains many fine features in programming. The first of these features is its built in functionality. The language is based on using symbolic data rather than performing numerical computations. Linked list is used as fundamental data structure. The list is a group of facts or data, connected and grouped into one structure. Another feature lisp has that helps in A.I. programming is its inherent relation to recursion.

LISP's syntax and semantics are derived from mathematical theory of recursive functions. This allows lisp to handle recursion with ease and any programmer the freedom to implement any function with recursion without worrying about lisp's capabilities. Lisp is an example of a procedural language. There is no main function for the computer to access, just one procedure after another. In these procedures, the programmer describes how to perform an algorithm. When accessed, this will give

the computer step by step instructions on "how to do" something, depending on the problem.

Most important point about lisp is its list-handling techniques. Lisp provides the programmer with powerful linked pointer structures. These pointer structures are all internally implemented with little to no dynamic memory allocation. The access functions to the structures are easy to use and internally defined. This leaves the programmer with absolutely no pointer management.

Characteristics of functional languages

1. Functions as first-class values
2. Programming with higher-order functions
3. Strong typing and static type checking
4. Polymorphism
5. Powerful module systems for large-scale programming
6. Recursive programming
7. Algebraic data types and pattern matching
8. Automatic garbage collection

(*iii*) The Data base Languages : The properties that distinguish languages designed to deal with databases are persistence and the management of change. Database entities do not disappear after a program terminates, but live on indefinitely as originally structured. Since the database, once organized, is permanent, these languages must also support change. The data may change and so too may the relationships between data entities or objects.

A database management system includes a **data definition language (DDL)** for describing a new collection of facts, or data, and a **data manipulation language (DML)** for interacting with existing databases.

A database can be viewed in several ways as shown in figure 10.5.

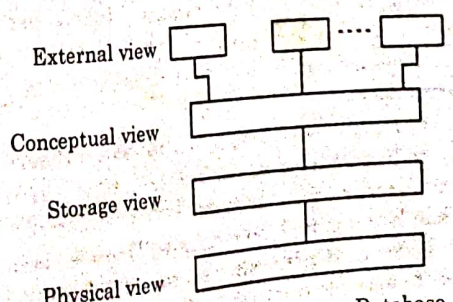


Fig. 10.5. Levels of Obstraction in a Database System

At the lowest level is the physical view, which describes the actual physical disks or drums where data is stored. At the next higher level of abstraction is the storage view, which gives structure to the physical data itself. The next higher abstraction is the conceptual view, which describes how the data is organized. Finally, there are possibly several external views to a database. These views are seen and used by the user, often through the query language. Taken together the host language plus the DSL, and the external, conceptual storage and physical views, comprise a database management system or DBMS. The example of database languages includes SQL.

10.3 STRUCTURED PROGRAMMING LANGUAGES

A program is structured if the flow of control through the program is evident from the syntactic structure of the program text. The structure of the program text should help us understand what the program does. The term is used for program design that emphasizes,

(i) Single-entry-single-exit program structure.
(ii) Hierarchical program structures design using only the simple control forms of composition, alternation and iteration.
(iii) Minimal use of **goto** statements.
(iv) Representation of the hierarchical design directly in the program text, using the structured control statements.
(v) The program text in which the textual sequence of statements corresponds to the execution sequence.
(vi) Use of single-purpose group of statements, even if statements.
(vii) The program is much easier to understand, debug verify to be correct, and later modify and reverify.

The structured languages are much more flexible than procedural languages. It is designed for programmer flexibility than absolute run time performance. Such languages have activation records but employ scope rules and nested block structure for storing and accessing data. They may require partial run time descriptors for structured data. Parameters may be transmitted either by value or by reference subprograms may be passed as parameters. During translation, static type checking for almost all operations is possible, so that little dynamic checking is required.

During execution, a central stack is used for subprogram activation records, with a heap storage area for data objects directly for use with pointer variables, and a static area for subprogram code segments and run time support routines.

Statement wise sequence control is based on structured control statements – compound statements, conditional and iterative. Subprograms are invoked with the usual call-return structure with recursion. The examples of structured programming languages includes Pascal, C and Ada etc.

10.4 COMPARISON OF DIFFERENT PROGRAMMING LANGUAGES

Here in this section we present the comparison of different programming languages.

10.4.1 PROCEDURAL AND NON-PROCEDURAL LANGUAGES

The differences are shown in table 10.1.

Procedural Languages	Non procedural languages
1. The procedural languages are command driven or statement oriented.	1. The non-procedural languages are fact oriented.
2. The programs in procedural languages specify what is to be accomplished by a program and instructs the computer on exactly how the computation is done.	2. The programs is non-procedural languages specify what is to be done and do not state exactly how a result is to be computed.
3. Procedural languages are used for application and system programming.	3. Non procedural languages are used in RDBMS, expert systems, natural language processing and education.
4. It is for complex.	4. It is for simpler then procedural.
5. These are basically imperative programming languages.	5. These are basically declarative programming languages.
6. The textual context or execution sequence is considered.	6. There is no need to consider textual context or execution sequence.
7. As an example of a program, the sorting is done by explaining in a C + + program all of the details of some sorting algorithm to a computer having C + + compiler. The computer, after translating the C + + program into machine	7. In a non-procedural language, it is necessary only to describe the characteristics of the sorted list. From this description, the non-procedural language system could produce the sorted list.

Procedural Languages	Non procedural languages
code or some interpretive intermediate code, follows the instructions and produces the sorted list.	
8. Machine efficiency is good.	8. The logic programs that use only resolution face serious problems of machine efficiency.
9. The procedural paradigm leads to a large no. of potential connections between functions and data, if there are many functions and many global data items.	9. There are no such connections present in non-procedural paradigm.
10. The examples of procedural languages are C, Ada, Pascal, C + + etc.	10. The examples of non procedural languages are Prolog, USP, Sql. scheme etc.

Table 10.1 : Differences between procedural and non-procedural languages

10.4.2 IMPERATIVE AND FUNCTIONAL LANGUAGES

The differences between Imperative and functional languages are given in table 10.2.

Imperative Languages	Functional languages
1. Imperative languages are based on Von-Neumann Architecture.	1. The function languages are not based on Von-Neumann architecture.
2. The programmer is concerned with management of variables and assignment of values to them.	2. The programmer need not be concerned with variables, because memory cells need not be abstracted into the language.
3. The imperative languages facilitate the computation by means of the state changes.	3. The functional languages facilitate the functions that the program represents, rather than just to state changes as the program executes, statement by statement.

Imperative Languages	Functional languages
4. Increased efficiency of execution.	4. Decreased efficiency of execution.
5. Laborious construction of programs.	5. Less labour required than programming in an imperative languages.
6. Very simple syntactic structure.	6. Much more complex syntactic structure than imperative language.
7. Concurrent execution is difficult to design and use.	7. Concurrent execution is easy to design and use.
8. The semantics are difficult to understand.	8. The semantics are simple as compared with Imperative languages.
9. The programmer must make a static division of the program into its concurrent parts, which are then written as tasks. This can be a complicated process.	9. Programs in functional languages can be divided into concurrent parts dynamically by the execution system, making the process highly adaptable to the hardware on which it is running.
10. The examples of Imperative languages includes C, C++, Ada, Pascal etc.	10. The examples of functional languages includes LISP, ML, scheme etc.

Table 10.2 : Differences between Imperative and Functional languages

10.4.3 C AND C + +

C is a procedural language. That is each statement in the language tells the computes to do something. C++ is derived from C language. It is a super set of C. Almost every correct statements in C is also a correct statement in C++, although reverse is not true. C++ is an object oriented language. Earlier C++, was known as C with classes. The major change was the addition of classes and a mechanism for inheriting class objects into other classes. Most C programs can be compiled by a C++ compiler, although there are a few inconsistences between the languages. Primitive data in C++ is the same in C. C++ expressions are the same as C. Relation between these two languages is shown in the figure.

The differences between C and C++ are given in table 10.3.

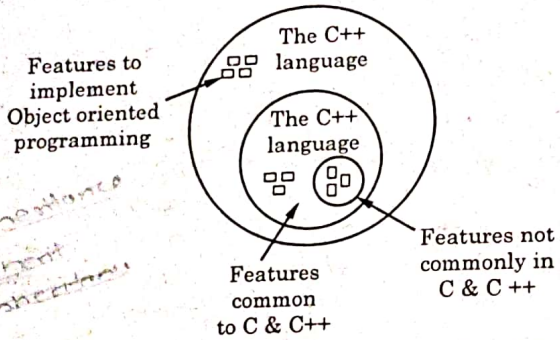


Fig. 10.6 The Relationship Between C and C++.

C	C++
1. C is a procedural language.	1. C++ is an object-oriented language.
2. C has no specific input and output statements.	2. C++ has specific input and output statements such as C in and C out.
3. It supports built-in and primitive date types.	3. It supports both built in and user defined data types.
4. No inheritance possible in C.	4. Inheritance is possible in C++.
5. There are no classes in C.	5. It supports classes, as it is a object oriented language, and object are instances of the classes.
6. Mapping between data and function is difficult and complicated.	6. Mapping between data and function can be used using **'object'**.
7. No virtual functions are present in C.	7. There are virtual function in C++.
8. It cannot be used to model a real world objects.	8. It can be easily used to model a real world object.
9. In C, we can call main ( ) function through other function or Main function could be recursive.	9. In C++, we cannot call main ( ) function through other or It can't be called recursively.

C	C++
10. In C, you have to declare variables at the start.	10. In C++, you can declare variables at any point with in a block not just at the beginning.
11. No namespaces are present in C.	11. Namespaces are present.
12. In C, polymorphism is not possible.	12. Polymorphism is major feature in C++.
13. Operator overloading is not possible in C.	13. Operator overloading is possible.
14. The function malioc ( ) and calloc ( ) are used for memory allocation.	14. The function New and Delete are used for memory allocation.
15. Sequence control of statements is done through expressions and statements such as compound statements, iteration statements etc.	15. Sequence control is done in similar may except handling through try and catch statement.
16. It supports top down parsing.	16. It supports bottom of parsing.
17. In C, if return type or data of parameters is not declared than those are integer by default.	17. If return type or data of parameter is not given it is not necessary that they are integer.
18. Multiple declaration of global variables are allowed.	18. Multiple declaration of global variables are not allowed.

Table 10.3 : Differences between C and C++

10.4.4 C++ AND JAVA

C++ was designed as an extension of C. C++ is an object oriented language. There is no features in C++ that directly corresponds to a java package. The closest similarity is a set of library functions that use a common header file. The Java interface is somewhat similar to a C++ abstract class. The main differences are as given in table 10.4.

JAVA	C++
1. Java has streamlined approach to memory allocation. It supports only NEW keyword.	1. C++ supports the NEW and Delete keyword for memory allocation.
2. The precise syntax of a java main function is slightly different. The syntax is :–	2. The declaration of main function is done at the starting and later on in the program

JAVA	C++

public static void main ( )
{
.....
}
	main ( )
	{
	.....
	}

3. Java does not include structures or unique.	3. It includes structur unique.
4. Java does not include pre-processor directive.	4. It includes pre directive.
5. Java does not perform any automatic type conversions.	5. C++ can do automa conversion.
6. Allows call, through the JAVA NATIVE INTERFACE.	6. Allows direct calls to system libraries.
7. No native support for unsigned arithmetic.	7. It supports native arithmetic.
8. It includes automatic garbage collection.	8. In C++, explicit management, garbage optional via-library.
9. Parameter can be passed only by value.	9. Parameters can be passe by value or by references.
10. Built-in data types are of specified size and range is defined by virtual machine.	10. In C++, a minimal r values is defined for buil types.
11. Java does not have pointers, it only has object references and array references.	11. In C++, pointer c manipulated directly as address value.
12. Strings in Java are objects and they cannot be terminated by a null character.	12. In C++ strings are ar character and can be ter by a null character.
13. It does not support multi-dimensional arrays.	13. It supports multi dim arrays.
14. Inline functions are not present.	14. Inline functions are pr C++.
15. There is no scope resolution operator.	15. There is scope r operator in C++.
16. Java is inherently multi threaded.	16. C++ is not inherentl threaded.
17. Java is platform independent.	17. It is not platform indepen
18. Java is an internet programming language.	18. C++ is not an programming language.
19. Java does not support goto statement.	19. C++ supports goto statem
20. It does not support type def.	20. It supports the keyword ty

JAVA	C++
21. It does not allow default arguments.	21. It allows default arguments.

Table 10.4 : Differences between C++ and Java.

10.4.5 C AND JAVA

The differences are given in table 10.5.

JAVA	C
1. Java is object oriented language.	1. C is procedural language.
2. The precise syntax of a Java main function is slightly different. The format is :– **public static void main ( )** { ..... }	2. The declaration of main is done at the starting or can be done later on in the program. **main ( )** { ..... }
3. Java is a robust language because, firstly there is no pre-processor or operator overloading.	3. C has a pre-processor without pre-processor programmes can't open header files.
4. Pointers are not used in java.	4. Pointers are used in C.
5. Java does not define the type modifiers auto, extern, registors, signed and unsigned.	5. In C language, the type modifiers auto, extern registors, signed and unsigned are used.
6. Strings in Java are objects. They are not terminated by a null character.	6. Strings in C are arrays of characters, terminated by a null character.
7. Java does not support multi-dimensional arrays.	7. C language supports multi-dimensional arrays.
8. We use classes in Java so polymorphism is present in Java. There is no multi inheritance in Java.	8. There are no classes in C language.
9. Java is an excellent garbage collector.	9. C is not an excellent garbage collector as compared to Java.
10. Java is platform independent.	10. C is not platform independent.
11. Java is an internet programming language.	11. C is not an internet programming language.
12. Java programs falls into two categories applications and applets.	12. The C programs do not fall in many categories.

Table 10.5 : Differences between C & Java

10.4.6 JAVA, C AND C++

	JAVA	C	C++
	terminated by a null strings can be concatenated using + operator.	terminated by a null character.	terminated by a null character.
12. Arrays	Arrays are quite different in Java. Array boundaries are strictly enforced. One array can be assigned to another in Java. Java does not support multi dimensional arrays.	C language supports multidimensional arrays.	C + + also supports multidimensional arrays.
13. Control flow	There is no goto in Java. The test expressions for control flow constructs return a boolean value in Java.	C doesn't uses goto statement. The test expressions for control flow constructs return an integer value.	C++ uses goto statement. The test expression for control flow constructs return an integer value.
14. Command Line Arguments	The command line arguments passed from the system into a Java program. In Java a single argument containing an array of strings is passed.	In C, two arguments are passed. One specifies the no. of arguments and other is a pointer to an array of characters containing the actual arguments.	The arguments are same defined as in C language.

Table 10.6 : Differences between C, C++ and Java

KEY POINTS TO REMEMBER

- The **programming languages** are used to facilitate the communication about the task of organizing and manipulating information and to express algorithms precisely.
- The programming language paradigms can be classified into :
  - — **Imperative programming languages**
  - — **Declarative programming languages.**
- The **Imperative paradigms** facilitate the computation by means of state changes.
- The imperative programming languages can be :
  - — **Procedural languages** *e.g.* C, Pascal, Fortran etc.
  - — **Object-oriented languages** *e.g.* C++, Java etc.

  - — **Parallel processing** facilitate the writing of declarations or truths (also known as facts).
- The declarative programming languages can be :
  - — The **logic programming languages** *e.g.* PROLOG
  - — The **functional programming languages** *e.g.* LISP
  - — The **Database programming languages**, *e.g.* SQL
- The **structured program** have single-entry-single -exit program structure.
- The three control structures used in structured programs are **composition**, **alteration** and **iteration**.
- The **procedural languages** are **command–driven** (or statement oriented) while the **non-procedural languages** are **fact–oriented**.
- The different programming languages can be compared on the basis of futures prevent in different programming languages.
-

EXERICISE

1. Define the term programming language. Explain different programming language paradigms with the help of a diagram in detail.
2. What are imperative and non-imperative (declarive) languages ? Explain in detail.
3. Differentiate between procedural and non-procedural languages in detail.
4. What are object-oriented languages ? Explain in detail.
5. What are functional languages ? Explain in detail.
6. What are logical languages ? Explain in detail.
7. Write short notes on :
   (*a*) C and C++
   (*b*) C and Java
   (*c*) C++ and Java
   (*d*) C, C++ and Java

PROGRAMMING LANGUAGES

Time : 3 Hours M.M. 100

UNIT-I

1. What do you under... by elementary data-types ? Discuss t... implementation of i... nd floating point real numbers.
2. (*a*) What makes ... guage ? Explain in detail.
   (*b*) Differentiat... dynamic type checking.
   (*c*) Write shor... pe conversion and coercion.

UNIT-II

3. (*a*) Discuss t... ion and implementation of structured t...
   (*b*) Give th... ormula for computing the location of c... matrix... :
   A : A... , $LB_2..UB_2$] where a stand and column-ma...
   (*c*) Diff... een array and record data structure.
4. Write sh... lowing :
   (*i*) In... ng (*ii*) Generic Subprograms
   (*iii*) S... subprogram (*iv*) Implementation of sets.

UNIT-III

...ollowing :
...t and explicit sequence control (*ii*) Actual and fo...
...implementation of recursive subprograms.
...fe... hods for transmitting parameters ? E...

UNIT-IV

...rocedural languages in detail.
...languages ? Explain in de...
...++ languages ? Explain th...

...atic Storage Manageme...

LANGUAGES

...T-I
...guages ?
...entary data type.
...g and Type Conversion ?
...mantics ? Explain.

...-II
... objects ? Explain th...
...s specification syntax a...

...II
... controlling the sequenc...
...iring storage during prog...
...n detail.
...ava.

...guages (*b*) Imperative and fu...